IMPROVEMENTS IN DISTRIBUTION OF METEOROLOGICAL DATA USING APPLICATION LAYER MULTICAST

A Thesis

by

SAURIN BIPIN SHAH

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2006

Major Subject: Computer Science

IMPROVEMENTS IN DISTRIBUTION OF METEOROLOGICAL DATA USING APPLICATION LAYER MULTICAST

A Thesis

by

SAURIN BIPIN SHAH

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Scott Pike
Committee Members,	Riccardo Bettati
	Evan Anderson
Head of Department,	Valerie Taylor

December 2006

Major Subject: Computer Science

ABSTRACT

Improvements in Distribution of Meteorological Data Using Application Layer Multicast. (December 2006)

Saurin Bipin Shah, B.E., Mumbai University

Chair of Advisory Committee: Dr. Scott Pike

The Unidata Program Center is an organization working with the University Center for Atmospheric Research (UCAR), in Colorado. It provides a broad variety of meteorological data, which is used by researchers in many real-world applications. This data is obtained from observation stations and distributed to various universities worldwide, using Unidata's own Internet Data Distribution (IDD) system, and software called the Local Data Manager (LDM).

The existing solution for data distribution has many limitations, like high end-to-end latency of data delivery, increased bandwidth usage at some nodes, poor scalability for future needs and manual intervention for adjusting to changes or faults in the network topology. Since the data is used in so many applications, the impact of these limitations is often substantial. This thesis removes these limitations by suggesting improvements in the IDD system and the LDM. We present new algorithms for constructing an application-layer data distribution network. This distribution network will form the basis of the improved LDM and the IDD system, and will remove most of the limitations given above.

Finally, we perform simulations and show that our algorithms achieve better average end-to-end latency as compared to that of the existing solution. We also compare the performance of our algorithms with a randomized solution. We find that for smaller topologies (where the number of nodes in the system are less than 38) the randomized solution constructs efficient distribution networks. However, if

the number of nodes in the system increases (more than 38), our solution constructs efficient distribution networks than the randomized solution. We also evaluate the performance of our algorithms as the number of nodes in the system increases and as the number of faults in the system increases. We find that even if the number of faults in the system increases, the average end-to-end latency decreases, thus showing that the distribution topology does not become inefficient.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Dr. Scott Pike and Gerry Creager for guiding and motivating me to give my best. Without their encouragement and support, this work would not have been possible. I would like to thank my committee members, Dr. Riccardo Bettati and Dr. Evan Anderson, for their willing assistance. I would also like to take this oppurtunity to thank Aparna Kanungo, Kaustav Ghoshal, Rohit Patil, and Gaurav Doshi for providing a lot of useful tips and feedback on my work. I also thank all those who took time off to proofread my thesis and give valuable suggestions.

TABLE OF CONTENTS

CHAPTER Page

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

A. Background

The Unidata Program Center [30] is an organization working with the University Corporation for Atmospheric Research (UCAR) in Colorado. Unidata provides a broad variety of data for use in geoscience education and research. The data are used in various applications like Environmental Prediction, Aircraft Control Systems and Weather Forecasting. As of 2006, Unidata provides more than 30 types of data-products [32] which contain information such as wind profiles, weather warnings, agricultural forecasts and national weather conditions. The Unidata community consists of nearly 150 universities; more than 50 GB of data is distributed among these universities everyday.

The Unidata community uses an architecture called the Internet Data Distribution (IDD) system [26] to distribute data over the Internet. The IDD system is a distributed network through which the users (universities) share meteorological data. Each university uses software called the Local Data Manager (LDM) [25] to control the data distribution mechanism. It is built on top of the IDD system, and provides programs that select, capture, manage, and distribute data-products through the IDD system. The IDD system can be compared to a network of aircraft routes, and the LDM can be compared to the air-traffic controllers at the airports. Figure 1 illustrates this data distribution mechanism using the IDD system and the LDM software. This thesis aims at improving methods of data distribution through the LDM and the IDD system.

This thesis follows the style of *Algorithmica.*

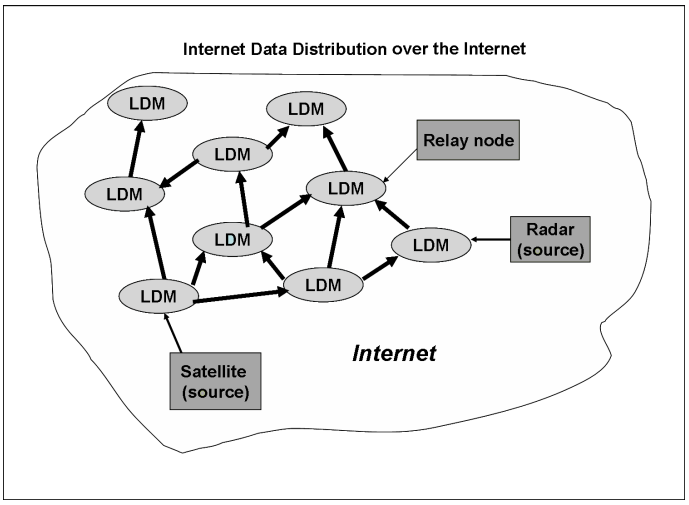

Fig. 1. The IDD System and the LDM

B. Motivation

The data distributed by the LDM through the IDD system are used to solve various real-world problems in disciplines such as hydrology, oceanography, and meteorology. Some of these applications require fast data delivery (low latency), some stress low cost (minimum bandwidth usage), while others require reliable data delivery (high fault tolerance). The LDM transfers large amounts of data for these applications. The volume of data can be estimated from the fact that the LDM protocol is the fourth largest user of Internet2 (after HTTP, NNTP and SSH) [13][1] as of May 2006.

Unidata monitors several characteristics of the participating servers, such as the data transfer rate, volume of data sent and received, end-to-end latency of data reaching the servers, and the data-products required by each server. It also monitors the topologies used for distribution for each of the data-products. These measurements have provided useful information about the current deficiencies in the IDD system

[1]Search for Unidata LDM in Table 7. on the web page

and the LDM such as higher latency of data delivery, increased bandwidth usage at the nodes, unstructured topology and poor scalability for future needs. This thesis provides techniques for improving these data delivery characteristics.

C. Contributions

In this thesis, we design algorithms to construct a network topology for distribution of data-products. We use application-layer multicast (also called as overlay multicast) fundamentals to construct this network topology. We use a heuristic approach to construct this distribution network. To construct the topology, each server applies a heuristic function to make a decision based on local knowledge. This distribution topology forms the basis of the improved IDD system and the LDM. The distribution network removes most of the limitations of the exiting solution by providing characteristics such as improved end-to-end latency, better scalability and better fault tolerance to the new IDD system and the LDM. We prove the correctness of our algorithms. We also evaluate the performance of our algorithms and the heuristic used in the algorithms using simulations. The basis of performance is comparison with the existing solution as well as comparison with a randomized solution.

D. Thesis Outline

Chapter II discusses Unidata's IDD system and the LDM in detail. It also points out the limitations of both these systems. Chapter III elaborates on the impact of these limitations and the problems caused in the real-world applications. We present a survey of IP multicast and overlay multicast fundamentals in Chapter IV. We also propose how Overlay Multicast fundamentals can be used to solve our problems. Chapter VI presents new algorithms to construct overlays that will provide efficient

means of data distribution, thus overcoming the limitations of the existing solution. Chapter VII gives the proof of correctness of our algorithms. Chapter VIII gives the simulation results and performance evaluation of our algorithms. Conclusions based on the simulation results and avenues for future work are presented in Chapter IX.

CHAPTER II

LOCAL DATA MANAGER AND INTERNET DATA DISTRIBUTION

A. Local Data Manager (LDM)

The LDM is software built on top of the IDD system. It is used by each node for event-driven data distribution through the IDD system. The LDM package contains software to control how to select, receive, process, and distribute meteorological data. Various types of data like satellite imagery data, national lighting data, and gridded binary data are contained in binary files called *data-products* which are then distributed using the LDM. The data-products are passed to the LDM in two ways: through *ingesters* [27] and through other upstream servers. Ingesters are programs which obtain data from observation stations, create data-products, and insert them into the LDM. The upstream servers are the data sites which send data-products to the LDM. After insertion, these data-products are stored in a temporary memory-mapped file called a *product-queue* [27]. These data-products can be extracted from the product-queue and sent to downstream servers which require the data-products. The LDM also provides a mechanism for local disposition of data-products by local processing, done using the decoder programs in the LDM package.

1. LDM Runtime Structure

The LDM runtime structure is shown in Figure 2. The main process of the LDM package called the *rpmd* process (LDM server) is on the left top corner of the figure. This process drives all other processes of the LDM package. The ingester process and the product-queue are also shown in the figure. The receiving LDM is also called a *downstream LDM* process. This process is responsible for receiving the data-products

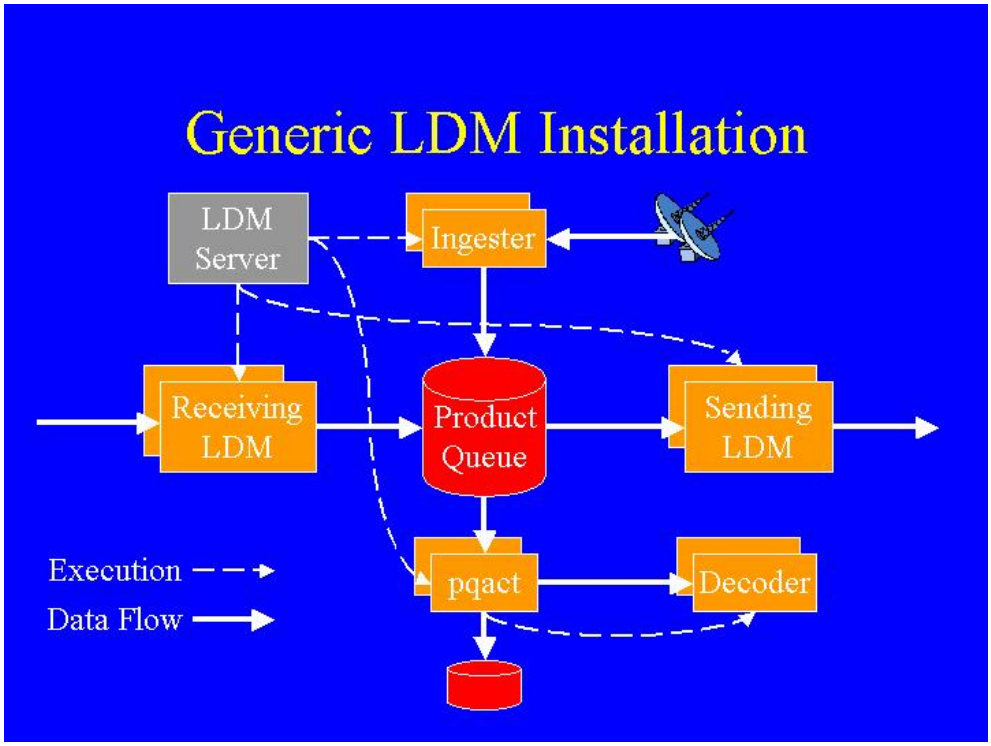

Fig. 2. The Generic LDM Installation and Runtime Structure [29]

from the sending LDM servers. The ingesters and the downstream LDM process get the data-products (either directly from observation stations or from other LDM processes as shown in Figure 2) and write them to the product-queue. Another process called *upstream LDM* (Sending LDM in Figure 2) sends data to other downstream servers and thus acts as a source or a relay node. The upstream LDM process extracts data from the product-queue and sends the data to other downstream sites. The data in the product-queue can also be processed locally. This task is done by the *pqact process*. This process extracts the products from the product-queue and writes them to either the disk or passes them to the decoder for further processing.

The LDM provides configuration files to perform the event-driven data distribution. If one server wants to request certain data-products from another server, or

wants to send certain data-products to another server, it can make these requests through configuration files provided by the LDM. For data transfer to occur between these servers, the configuration files of both servers must have entries for each other. Currently, these entries are updated manually by the site administrators. Thus, whenever a server goes down either temporarily (for maintenance) or permanently (crashes), all the servers which either receive data from or send data to this server need to change their configuration files. This process requires manual intervention by the administrators who monitor the LDMs. This static, hard-coded solution for data transfer is one of the biggest limitations of the LDM. This thesis provides means to automate these changes to the configuration files.

B. Internet Data Distribution (IDD)

The IDD system is a distributed system through which the cooperating universities distribute meteorological data amongst each other. It allows the participating universities to subscribe for certain data-products, which are then streamed to the university servers. The architecture of data delivery consists of two phases: collection of data from the observation stations, and distribution of data to the users. In this thesis we will focus on the second phase of the data delivery architecture.

Currently the IDD system is based on a hierarchical distribution scheme as shown in Figure 3. The figure shows the IDD topology for a sample data-product called CONDUIT. This data-product contains high-resolution data. The graph represents the latency reports on April 12th, 2006 at 15.38 GMT. The scale bar on the left hand side of the figure shows the latency in minutes associated with data delivery. All the links are uni-directional, and represent links from the sender to the receiver. Approximately 25–30 GB of data is transferred everyday over the CONDUIT network.

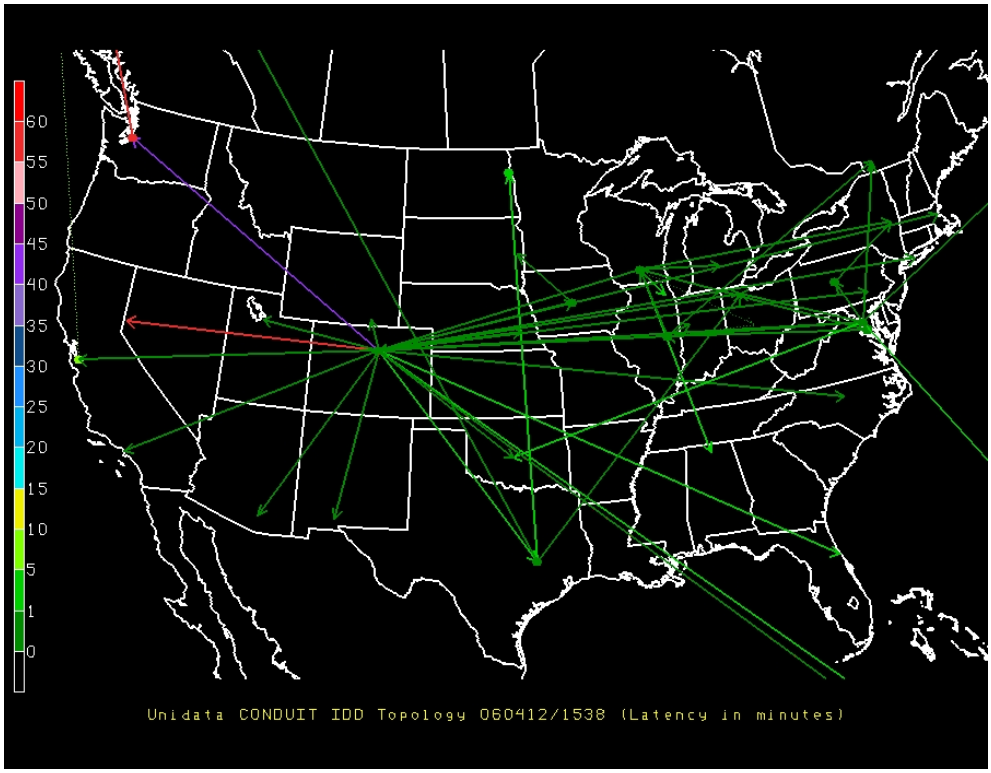

Fig. 3. CONDUIT IDD Topology [28]

The topology consists of two types of nodes (sites): data sources and data relays. Each new server that joins the IDD system, is expected to act as a relay node whenever required. The relay nodes *push* data downstream as soon as it is received. This is called *Push Technology*, and it enables new users to be added to the distribution topology, without increasing the load on the limited number of upper level servers.

The site administrators of the relay nodes have several responsibilities, like monitoring the IDD system, addressing problems of downstream sites, and responding to topology changes as necessary. Not all site administrators have the time, personnel, hardware or technical knowledge to perform such duties [1]. So, although all the sites are expected to act as relay nodes, all sites are not capable of performing such duties.

Thus, there is often a shortage of relay nodes in the IDD system to send data and handle the addition of new leaf nodes.

C. Limitations of LDM and IDD

As mentioned in Chapter I, Unidata measures several data-delivery characteristics of the IDD system. These measurements provide useful information about current deficiencies in the IDD system and the constituent LDMs. We discuss these limitations to better understand the impact of these deficiencies and to motivate the aim of this thesis.

Manual Intervention: In Section A, we explained why manual intervention is required to transfer data between two servers. If the topology changes, then the configuration files also need to be changed. Manual intervention is required to make these changes. This is one of the major limitations of the LDM which needs to be addressed. Ideally, each LDM should have the ability to automatically configure itself to get data from the best connection, as stated in [1].

Scalability: Section B talks about the shortage of relay nodes to handle the addition of new leaf nodes to the IDD system. This limitation reduces the scalability of the IDD system with respect to adding new users as leaf nodes to the IDD topology.

Latency: The IDD system distributes large volumes of data everyday. Considering the amount of data, sometimes it takes a long time to transfer the data. In the existing solution, some nodes receive data with a very high end-to-end latency (30–40 minutes). Such data might be stale and may not be useful for time-sensitive applications, such as weather forecasting.

Bandwidth: Site Bandwidth consists of two parts: *in-bandwidth* is the bandwidth consumed by the data flowing into the node and *out-bandwidth* is the bandwidth consumed by the data flowing out of the node. Some nodes in the IDD topology send the same data streams to many children. As shown in Figure 3, the Unidata site in Colorado sends data to many children and therefore the out-bandwidth usage of this site is very high. For many upper level servers in the IDD topology, the out-bandwidth usage is approaching capacity because of the growing number of children.

Decentralized and Unstructured Topology: The IDD topology is a type of peer-to-peer topology. In [16], the authors classify peer-to-peer topologies into three types: *centralized*, *decentralized but structured*, and *decentralized as well as unstructured*. In a centralized topology, there is a central authority that controls and knows the properties of all other nodes in the system. Decentralized but structured topologies do not have a central controlling authority, but these topologies are characterized by a tightly-controlled structure (connections between the nodes). The nodes in the topology are not placed arbitrarily, but are placed at locations based on their properties. In the decentralized as well as unstructured topology, there is no control over the topology. In such a topology, the placement of nodes is not based on the topology knowledge.

The IDD system has a decentralized as well as unstructured topology. Due to loose control over the placement of nodes, there is little scope for optimization of characteristics such as end-to-end latency of data reaching the servers, bandwidth (both in-bandwidth and out-bandwidth) consumption and fault-tolerance with respect to node crashes.

In the next chapter, we will discuss in detail, the problems caused due to these limitations. We will also discuss the impact of the problems in the real world.

CHAPTER III

PROBLEMS FACED DUE TO LIMITATIONS OF IDD AND LDM

Chapter II discussed some limitations of the LDM and the IDD system regarding data distribution. Since the data distributed using the LDM and the IDD system are used by a large number of real-time applications, these limitations create many problems, some of which are mentioned below.

A. Need for Automated Topology Decisions

Stephen Chiswell in [4] suggests the need for the IDD topology decisions to be automated. Currently the IDD topologies are maintained based on a site's data requirements and the ability of the upstream servers to deliver the requested data. To deliver this data, the administrators of the upstream sites need to take on the responsibility of monitoring the IDD system and the LDM. Moreover, they have the added responsibility of addressing the problems of downstream servers and responding to topology changes. Not all site administrators have the time, personnel, network connectivity, hardware or technical knowledge to execute such responsibilities [1]. Local network conditions may also necessitate a site to request some other upstream servers to satisfy its data requirements, thus causing a change in the topology. When topology becomes increasingly dynamic, it becomes difficult to deal with the topology changes manually. For example, manually dealing with issues such as a site being promoted to a higher tier in the distribution tree, takes time and a large amount of coordination [1].

B. Fault Tolerance

We consider two types of faults in the current IDD system: permanent crashes and temporary crash-recover faults. The first type of fault occurs in cases such as power failures and hard disk crashes. In this case, the node is removed from the topology permanently and it may join the system later only as a new node. The second type of fault occurs when a server goes down temporarily for maintenance. In this case, the node joins the topology as the same node and in the same position later on. These faults are common among most of the data relay sites. Since the data are used in various real-time applications, reliability of data transfer (data completeness) is an important factor.

Whenever a fault occurs, the downstream sites need to receive data from some other upstream sites. For this purpose, all participating sites within the IDD system are assigned *fail-over upstream relays*. Fail-over upstream relays are the data sites which send data to downstream nodes, if the downstream nodes fail to receive data from their parents in case of faults. However, unless configured manually, the downstream servers cannot start accepting the data from the fail-over relays.

In case of planned outages, (server goes down for scheduled maintenance) the downstream severs can be pre-configured to start accepting data from upstream relays at proper time, but in case of unplanned faults (permanent crashes), there is a significant delay between the time at which fault occurs and time at which new links are created, thereby resulting in data loss. Many applications like Aircraft Control Systems and Weather Reporting Systems require high reliability of data with respect to completeness. Hence, the data distribution mechanism must have good fault-tolerance characteristics to ensure smooth and reliable data transfer.

C. Increasing Data Types

Each data-product in the LDM is called a *Feed type*. The existing solution for data distribution creates different topologies for different feed types. All these topologies are part of the IDD system and they need to be monitored continuously. Currently there are 30 feed types associated with the LDM and with ever-increasing volumes and varieties of data, this number will surely increase.

D. Large Hardware Requirements

Increases in both data volume and the number of participating sites are pushing the LDM and the management of the resulting topologies to their limits. As mentioned in Chapter II, each LDM server receives data-products from an upstream server, stores them in the product-queue, and then sends them to downstream servers. It is recommended for each LDM server to store at least one hour's worth of data in the local product-queue. The immediate impact of increasing data volume is the increase in product-queue size of the each relay server [31]. As stated in [31] most of the sites in 2002 had already reached their maximum queue size of 2 GB. As a result, many upper level servers (for example, Unidata's own sever) started rebuilding the LDM with large queues. Building large product-queues requires the servers to have large file support so that they can support 64 bit addresses [31]. All servers do not have this capability, but it is becoming necessary for most of the top level servers to support such large hardware requirements.

E. End-to-End Latency of Data

Unidata provides statistics [28] of the topologies of all the data-products and the participating nodes. These statistics include the network topology for each data-product,

the end-to-end latency of data reaching the servers, the data-products handled by each server, the volume of data sent and received by each server, and the data-product paths from source to each server. These statistics provide a basis for developing and testing algorithms to detect trouble spots in the data distribution topologies. Using these statistics, we have found that the end-to-end latency of data for some servers is as high as 30–40 minutes. Most of these data are used for Environmental Prediction and Severe Weather Detection. Such applications require up-to-date data with minimum latency. Hence, the data distribution mechanism must provide means to deliver the data with minimal latency possible.

Since the data distributed by the LDM and the IDD system are used in many real-time applications, the above mentioned problems can have undesirable impacts. In this thesis, we intend to solve these problems by constructing better data distribution topologies. In the next chapter, we will discuss the fundamentals of multicast, and methods of implementing these fundamentals. We will also propose to use multicast fundamentals to create a data distribution network which will solve most of the problems mentioned above.

CHAPTER IV

MULTICAST FUNDAMENTALS AND OVERLAY MULTICAST

In Chapter II, we talked about the limitations of the Unidata LDM and the IDD system. We also discussed the problems caused due to these limitations in Chapter III. To solve these problems, we need to construct an efficient and reliable distribution network for data delivery to users. In this chapter, we will look at *multicast* and *overlay multicast* fundamentals to construct such a distribution topology. These fundamentals are explained in the subsequent sections.

A. Multicast Overview

Multicast is a technology through which a single stream of data can be simultaneously sent to several users. Steve Deering first introduced multicast fundamentals in 1989, and since then it has been used for many applications such as video conferencing, stock quotes, radio and television broadcasting, news and distance learning. Multicast delivers the data from the source to potentially thousands of users without increasing the load on the source or the receivers. Deering proposed that multicast should be implemented at the network layer by a mechanism called *IP Multicast* [6]. Figure 4 illustrates the IP multicast mechanism. Users located anywhere on the Internet can form a multicast group to receive multicast data. The source then sends data to this multicast group to deliver it to each of the group members.

1. IP Multicast Fundamentals

An important aspect of IP Multicast is the identification of the receivers. Each user who wishes to obtain the data joins a group called a *Multicast Group*. Each multicast group dynamically obtains a unique Class D IP address. To deliver multicast

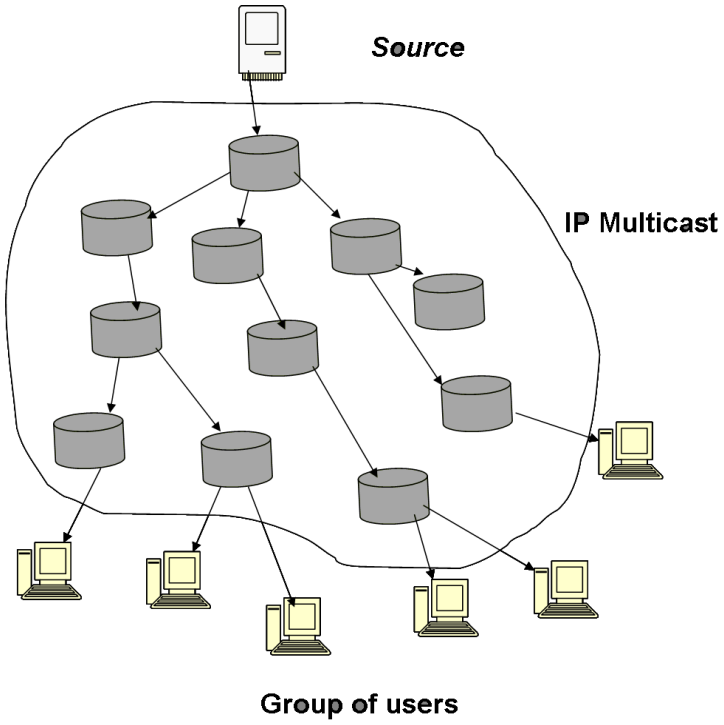

Fig. 4. IP Multicast

data to all the users, the source needs to send just a single stream of data to the unique IP address of the Multicast Group. This data is replicated and forwarded by the network routers to all the members (users) of the group. The dependency on the network routers follows from the fact that IP multicast is implemented at the network layer. Several multicast routing protocols, such as DVMRP [34], Protocol Independent Multicast (PIM) -Sparse mode [7], Multicast Extensions for Open Shortest Path First (MOSPF) [19], and others have been developed for data transfer using IP multicast fundamentals.

2. Problems with IP Multicast

IP Multicast provides an efficient mechanism for data transfer from the source to the group of receivers. Even though more than a decade has passed since IP Multicast was first proposed, it is not fully deployable in the Internet today. The deployment of IP Multicast poses many problems [5], some of which are listed below.

Dependency on Routers: IP Multicast depends on the routers, both the end routers as well as the intermediate routers. Not all routers in the Internet today support IP Multicast. Therefore, full deployment of IP Multicast has not yet been possible.

Poor Routing Scaling Capability: IP Multicast requires the routers to maintain *per group state i.e.*, routers need to keep updated information about each of the Multicast Groups. This introduces problems as the number of groups increases, resulting in poor routing scaling capability.

Difficulty in Supporting Higher-Level Functionalities: IP Multicast works at the IP or the network level. Therefore it is difficult to provide higher level functionalities such as reliability, congestion control, flow control, and security.

No Control over Data Transfer: IP Multicast allows an arbitrary user to send data to an arbitrary group. Some malicious users may exploit this functionality to flood or attack the network. This makes the network vulnerable and difficult to manage.

B. Overlay Multicast

Due to the several problems with IP Multicast, researchers [5], [2] [8] thought that the network layer is not necessarily the best layer to perform multicast. They pro-

posed that the multicast mechanism should be performed on higher layers such as the Transport Layer or the Application Layer. Such an Application Layer Multicast mechanism is called *Overlay Multicast*. The basic idea of this mechanism is to have the end-systems perform multicast rather than the routers. The data is replicated by the applications running on the end-hosts and not by the underlying network routers. Figure 5 gives an idea of how overlay multicast works.

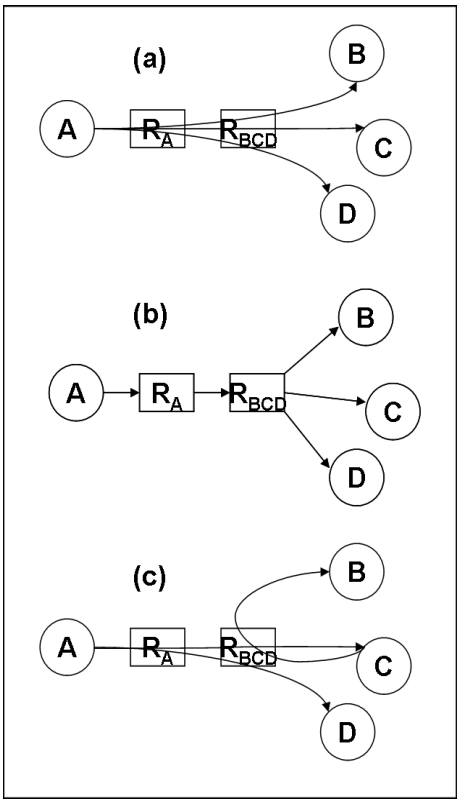

Fig. 5. (a)Unicast (b)IP Multicast (c)Overlay Multicast

The figure shows four nodes *A, B, C* and *D*. *A* is the source sending data, while *B, C* and *D* are the receivers. R_A is the router connected to *A*, while R_{BCD} is the router connected to the end hosts *B, C* and *D*. Panel (a) shows the simple unicast mechanism, where *A* sends three copies of data to all three receiving hosts.

This method ensures fast delivery, but it also consumes more bandwidth at A and is inefficient. Panel (b) shows IP Multicast. In this case, A sends a single copy of data for all of them. Router R_A forwards the data to the router R_{BCD}, which replicates the data and sends distinct copies of data to each of the receiving end hosts B, C and D. Panel (c) shows the overlay multicast mechanism. In this case, A sends two copies of data, one for C and one for D. The end-host C then does the replication of data and forwards it to B.

The most important advantage of overlay multicast is that it is fully deployable in today's Internet. Since the mechanism does not depend on the routers and depends only on the end-hosts, overlay multicast can be easily deployed now. The basic idea of overlay multicast is to form an overlay network of the end-hosts and place it over the current Internet. The end-hosts perform all the multicast activities like sending, receiving, replicating and forwarding data. Any two end-hosts are connected by unicast tunnels and perform all data transfer activities through these tunnels. Apart from being deployable, other advantages of this mechanism include no dependencies on routers, provision of higher-level functionalities, and robustness. Also the users of the overlay multicast mechanism do not need to join a group with a Class D IP address, which removes the necessity of dynamically obtaining a group IP address.

While overlay multicast provides some advantages, we need to consider other issues with this mechanism. We will discuss such issues in the next subsection.

1. Overlay Multicast Issues

While overlay multicast is a good alternative, it is not as efficient as IP Multicast. Also, there are some issues regarding scalability, lack of network information, performance, and robustness which are discussed below.

Comparison with IP Multicast: Overlay multicast is not as efficient as IP Multicast. In the Panel (c) of Figure 5, the data sent by the source A is sent twice in the link A-R_A in the overlay multicast scheme, as compared to just once in the IP Multicast Scheme. Since the overlays do not depend on the routers to replicate and forward the data, this cannot be avoided. Hence, in overlay multicast, the same data is sent over many links thus increasing the *link stress*. Link stress is defined as the number of times the same data is sent over the same link. The best we can do with overlay multicast is to make it as close to IP Multicast as possible considering the efficiency.

No Network Information: The end-hosts may not have information about the underlying network. Data duplication and forwarding is done at the application level and it is independent of the underlying network. Therefore, data may traverse the network many times before it reaches the end-hosts. This leads to an inefficient use of the network capability. This disadvantage can be removed if the overlay of the end-hosts can be built with the Internet topology in consideration.

Scalability Issues: Many factors can affect the scalability of overlays. If the overlays designed are such that they require to keep the complete topology information, then each node in the overlay needs to keep the information about all other nodes in the overlay. If the number of nodes in the overlay or system increases, then it is obvious that the amount of updated information (about the nodes in the topology) required by each node also increases. This might adversely affect the scalability of the system. Usually the nodes in overlays with high robustness keep information about all the other nodes, and this seriously affects the scalability. Scalability is also affected by the overhead caused by the control

messages in the system. If a lot of control messages are used, then the scalability of the system reduces.

Robustness: The robustness of the system depends on the amount of information each node keeps. If each node keeps information about all the nodes in the overlay then it helps the overlay in easily detecting the faults and propagating this information to all the member nodes. This gives a higher degree of robustness to the system. The robustness of the system can also be increased via data redundancy. The control messages in the system also help to increase the robustness. However, the increasing amount of information with each node and the increasing number of control messages reduces the scalability of the system. So a trade-off between scalability and robustness needs to be considered while constructing the overlay.

2. Types of Overlays

Many trade-offs are considered while developing overlays for an application. The overlays are designed according to the needs and characteristics of the application. Applications like video streaming for conferences require minimum latency, while some other applications require minimal cost by using less bandwidth. Some applications like stocks quotes or weather reports require high robustness, while others like broadcasting require high scalability. So researchers over a period of years [5] [8] [21] have designed different overlays for the different characteristics of overlay multicast desired. Different types of overlays are given below.

Tree-Based Overlays: Tree-based overlays generate a tree structure for data delivery to the end users. Some tree building algorithms use shortest path or minimum spanning trees [8], while other tree overlays use core-based trees [15].

Shortest-path trees are useful for one-to-many (one source to many users) multicast applications, while core-based trees are useful for many-to-many multicast applications. The HTMP (Host Multicast tree protocol)[36] is one protocol that uses tree based-overlays. The overhead caused by control messages passed in tree overlays is less because of the fewer number of links in the structure. The tree-based approaches are useful for achieving low ene-to-end latency, reducing cost due to bandwidth, and achieving better scalability. But these approaches are not as robust as the mesh-based approaches because a node failure in the overlay may cause the tree to be partitioned into two or more disjoint parts, thus disrupting the data distribution mechanism.

Mesh-Based Overlays: Mesh structures provide multiple *path-disjoint* routes from the root to the end users. Two routes which do not have dependencies on each other (*i.e.*, they are independent with respect to all the nodes in each path) are called path-disjoint routes. As a result, even if an intermediate node or link fails, there are alternate routes to the destination through which the data can be transferred. Overlays having a mesh-based structure are called Mesh Based Overlays. The Narada protocol [5] was one of the first protocols to give the feasibility of application layer multicast.

These type of overlays provide robustness to the system at the expense of other characteristics. For example, sending data on multiple paths increases data redundancy. Also, mesh overlays require more control messages to be sent, so the same data are transmitted over the same link several times, thus increasing link stress. Each node in a mesh also needs to keep more information, as the number of links in the mesh is more than a simple tree, thus reducing the scalability of the system. So, mesh-based overlays are useful for applications

where greater robustness is an important criterion.

Implicit Overlays: Some overlay protocols adopt an implicit approach where they create neither a mesh nor a tree, but rather the structure is created implicity using a data forwarding algorithm. These overlays have data forwarding routes based on the position and control topology of the nodes. The Content-Addressable Network (CAN) [21] protocol is an example of the implicit overlay protocol. In this protocol, the end-hosts implement a distributed hash table on an Internet wide-scale and form a distribution network. Suman Banerjee *et al.* in [2] proposed a scalable application layer protocol, called NICE. The protocol arranges the end-hosts in an hierarchical topology to perform data distribution. The implicit overlays are designed to be compatible with groups consisting large numbers of nodes.

C. Using Overlays to Solve Our Problem

We saw that IP Multicast is a very good way to perform data distribution from the source to a group of members. However, it is not deployable in the current Internet. Also, it has other problems which have been suggested in the previous sections. Although overlay multicast is not as efficient as IP Multicast, it is deployable currently and it serves as a good alternative to IP Multicast. We will use overlay multicast to solve the problems with IDD and LDM mentioned in Chapter III.

In this section we will suggest improvements to remove the limitations of the LDM and the IDD system by improving their performance characteristics. These characteristics include end-to-end latency of data reaching the nodes, bandwidth consumption at the nodes, improved fault tolerance with respect to node crashes (either crash-recover faults or permanent crashes), fairness with respect to the number of children,

and bandwidth usage at the nodes. To achieve this, we will construct an overlay data distribution network using overlay multicast fundamentals. The distribution network will be a decentralized but structured topology. Also, it will not require manual intervention for adapting to changes in the topology. We saw that the tree-based overlays are efficient for data distribution, but they have poor robustness characteristics. The mesh-based overlays on the other hand are not as efficient as the tree-based overlays, but have better robustness properties. Since many applications using meteorological data require high robustness, we will use mesh-based overlays for constructing our distribution network. Using this mesh, we will construct a spanning tree (a subset of the mesh) as a primary means of data distribution. This way, we will not lose the efficiency characteristics of tree-based overlays. The rest of the mesh will be used for data redundancy to improve robustness. Listed below are the characteristics which we intend to improve and the methods which we will need to follow.

Latency: The end-to-end delay of data at a node consists of two parts: delay caused by data collection from the observation stations and delay caused by data distribution to the end users. The delay due to data collection is beyond the scope of this thesis. Instead we will focus on measures to improve latency caused by the data distribution mechanism. In a broad overlay with less depth, data requires fewer hops to reach the destination nodes. Thus, a broad overlay helps in achieving low end-to-end latency.

Bandwidth: We need to construct a distribution mechanism such that the out-bandwidth usage will be distributed among all the nodes in the topology. In a deeper overlay, each node has fewer children, so less out-bandwidth is consumed in forwarding data. Thus, a deeper topology helps in reducing the out-bandwidth usage at the upper-level nodes, and in distributing the out-

bandwidth usage among many other nodes of the overlay.

While designing algorithms to construct the overlay distribution network, we need to consider the trade-off regarding the nature of the overlay. A deeper topology will reduce out-bandwidth usage at the few upper nodes, but data will take multiple hops to reach the lower-level nodes, thus causing higher latency. On the other hand, a broader topology will reduce the end-to-end latency at the nodes, but will increase the out-bandwidth usage at the upper-level nodes. Each node has a maximum out-bandwidth. We will design algorithms that will maintain a balance between out-bandwidth usage and end-to-end latency. The number of children and out-bandwidth usage of each node will be based on the maximum out-bandwidth capacity of that node.

Fault Tolerance: Mainly two types of faults occur within the servers in the current IDD system: permanent crashes and crash-recover faults. The first type of fault occurs in cases like power failures and disk crashes. In this case, the node is removed from the topology permanently, and it may join the system later only as a new node. The second type of fault occurs when a server goes off-line temporarily for maintenance. In this case, the node joins the topology as the same node and in the same position later on.

Since the data are used for various real-time applications, reliability of data transfer (data completeness) is an important factor. So, we must design algorithms such that each node in the overlay continues to receive data-products even after any of the above mentioned faults occur. In the previous section, we discussed the robustness properties of the tree-based and mesh-based overlays. Failure in a tree structure causes disruption in the data transfer. On the other hand, failure in mesh structures does not cause disruption in data delivery as

long as all the path-disjoint paths to a node are not cut because of the failure. We will exploit such characteristics of mesh-based overlays to achieve better fault tolerance.

Manual Intervention: We will design algorithms to construct the data distribution mechanism and then manage the topology. If changes occur in the topology, (*e.g.,* a node leaves the topology) the The algorithms will also adjust the links among the other nodes and thus adapt to the changes in the topology. It will also make automatic changes in the LDM configuration files, thus eliminating the need for manual intervention.

In the next chapter, we will discuss the algorithms to construct and manage the topology.

CHAPTER V

PROBLEM DEFINITION

In Chapter III, we discussed the limitations of the LDM and the IDD topology. In Chapter IV, we suggested methods for creating a data distribution overlay network. In this chapter, we will discuss in detail the challenges faced, the problem definition, and related work.

A. Challenges Faced

As we discussed in Chapter IV we will use overlay multicast fundamentals to construct an overlay distribution network. Some challenges that are faced while constructing the overlay are as follows.

- Since overlay multicast is implemented at the application level, the overlay construction is decided by the end systems and not by the underlying network routers. We will construct an overlay without complete knowledge of the network routers or the underlying network topology. The algorithm design is based only on the knowledge of the end hosts in the system and their properties.

- The end systems of the overlay topologies are connected by unicast channels. These unicast channels are never more efficient than the corresponding multicast channels, with respect to the end-to-end latency of data delivery, redundant traffic, and throughput. These deficiencies cannot be eliminated.

- If an overlay building mechanism requires complete knowledge of all nodes in the topology, then each participating node needs to keep updated knowledge of all the other nodes. This hampers the topology construction mechanism as the number of nodes increases. To construct overlays with good scalability we

need to design algorithms that depend only on local or partial knowledge of the topology. This decentralized approach based on limited knowledge makes tree generation more challenging, but more scalable.

- Out-bandwidth is a resource of each node in the topology, and it signifies the bandwidth consumed by data flowing out of the node. The number of data streams which a particular node can send to its children is bounded by the maximum value of its out-bandwidth. Since the out-bandwidth is finite, the fan-out of each node is also finite. This bound is important because it controls the traffic of the data flowing in its subtree. The challenge in constructing the overlay is that the overlay should be subject to the constraints regarding the bounds on the fan-out of the nodes in the topology.

B. Basic Definitions, Computational Model, and Problem Formulation

1. Basic Definitions and Assumptions

The underlying network consists of routers connected by links. We assume that it is a connected graph. We construct an overlay and place it on top of the underlying network. The overlay is constructed using a set of nodes, say S. The elements of S are the servers of the universities which require data from Unidata. The constructed overlay network is modelled by a graph $G = (V, E)$, where V is the set of vertices (the university servers) and E is the set of edges (links in the overlay connecting two servers). A source node $s \in V$ distributes data to the overlay network. Each node in V is an end system and it is also an element of the set S.

Definition B.1. *An edge $e \in E$ is represented by (i, j) and it denotes the bi-directional virtual link between the servers i and j in the overlay graph, where i,j are elements of V.*

Definition B.2. *The end-to-end latency of data for a node n is the time taken by data to reach n after it leaves the root.*

Definition B.3. *Average end-to-end latency is defined as the average of the end-to-end latency of data from root to all the nodes in the topology.*

Our goal is to obtain minimal end-to-end latency of data delivery at the nodes. So, the cost function of our topology is defined as the average end-to-end latency of data delivery to the nodes in the system.

Definition B.4. *Each node n which belongs to the set S has a resource R_n which is the out-bandwidth capacity of the node.*

This out-bandwidth resource is used by each node to send data to its children and backed children. Since the resource R_n is finite, the number of children nodes (fan-out) of n are also finite.

Definition B.5. *Each node n has a set DP_n which consists of all the data-products that node n uses.*

Suppose that the total number of data-products provided by Unidata is K. Then $|DP_n| \leq K$.

Definition B.6. *The fan-out of a node n, with resource capacity R_n is equal to the total number of children that the node n can send data-products.*

Even if a node requires $< K$ data-products, a parent always reserves bandwidth equal to the size of the global set of K data-products for sending data-products to its child. This surplus bandwidth is reserved for future needs. Let the size of the global set of K data-products be $Size_K$. Hence the fan-out of a node n can be calculated using

$$fanout_n = \frac{R_n}{Size_K} \tag{5.1}$$

From this definition, it is clear that the fan-out of each node depends on the resource capacity of that node (since the $Size_K$ is constant for all nodes). Hence, to construct a flat topology, the nodes with high resource capacities should be on the upper levels of the topology. As a result, we construct the topology with a constraint that "For each parent child pair (P, C), $R_P \geq R_C$".

Definition B.7. *A node n is said to have node-disjointness, if for each intermediate node i on the primary path to n, there exists at least one route ($Route_i$) from the root to n, such that $Route_i$ does not contain i.*

2. Computational Model

Our computational model is an asynchronous system subject to *crash faults*. When a process which was executing correctly, suddenly stops executing, a crash fault is said to have occurred [3].

This system is augmented with a *perfect failure detector*. A failure detector is a distributed oracle that is a module augmented to all processes, to detect crashed nodes (faults in the system). The perfect failure detector was first proposed by Chandra and Toueg in [3]. This failure detector has the following properties.

1. **Strong completeness:** For every run, eventually every process that crashes is permanently suspected by every other correct process.

2. **Strong Accuracy:** For every run, no correct process is suspected before it crashes.

We also assume that the nodes in the underlying topology are connected by *reliable channels*. The communication channels which guarantee that every message that was sent will also be delivered at its destination are called reliable channels.

These channels do not cause errors like loss of an expected message or delivery of an unexpected message at the receiver.

A failure detector is used for abstracting and separating the failure detection mechanism from the application. Using a failure detector allows us to design algorithms without worrying about lower-level network issues such as time-outs for fault detection. We do not discuss the implementation of the failure detectors in this thesis, but we use the properties of the failure detector in our algorithms.

A perfect failure detector can not be implemented in a purely asynchronous system. However, it can be implemented in a synchronous system, and stronger partially synchronous systems. We do not discuss the implementation of the perfect failure detector in this thesis. This failure detector abstraction allows us to design algorithms for our application and provide proofs of correctness in a very clean manner.

3. Problem Formulation

Given the assumptions made in the previous subsections, and the input information about the set S of nodes, the set DP_n of data-products with each node and the resource capacity R_n of each node, we need to design algorithms to construct trees such that:

1. All trees have the same root s. The trees are such that overlapping them creates a DAG structure.

2. For each parent-child pair (P, C), $R_P \geq R_C$

3. The out-bandwidth usage of any node n does not exceed its resource capacity R_n. Hence, the fan-out of any node n is bounded.

4. The average end-to-end latency of the DAG structure is minimal.

5. There are multiple routes from root to each node (except the direct children of the root) such that each node has node-disjointness.

A spanning tree which is a subset of the DAG structure, is used for primary data transfer, while other trees are used for back up data transfer.

The constructed DAG topology should have the following three properties for its correctness:

Safety: Whenever a new node is added to the topology, the node-disjointness of each node in the topology should be unaffected.

Progress: If a new node is added to the topology, eventually after a finite time interval, the new node should start receiving all the data-products it requires.

Tolerance: The topology should be able to tolerate a single node crash. A node crash causes many nodes in the topology to lose their node-disjointness. Consequently, rearrangements should take place in the topology and these rearrangements should restore the node-disjointness of all the affected nodes in the topology.

Our problem formulation concentrates mainly on three objectives: (1) constraining the out-bandwidth usage of the nodes (2) minimizing average end-to-end latency of data in the distribution network and (3) constructing multiple node-disjoint routes.

The dynamic version of the problem is the one that is stated above. However, we also define a static version of the problem. In the static version of this problem, we are given the overlay topology (DAG structure) and we need to construct a spanning tree subject to degree constraints of the nodes. The cost (average end-to-end latency of data) of the spanning tree should be as minimum as possible. As a standard for comparison, the cost of the spanning tree should be less than a pre-decided constant B.

It is difficult to prove the hardness of the dynamic problem, so we prove the hardness of the static problem. We have sketched a proof to show that this static version of the problem is NP-complete. The proof is using a reduction from a known NP-complete problem (Hamiltonian Path problem) to this problem. A Hamilton path is a path between two vertices of a graph that visits each vertex of the graph exactly once. Finding a Hamiltonian path in a graph is known to be a NP-complete problem [9].

The proof is as follows:

Lemma B.1. *The problem of constructing a spanning tree with minimum cost (less than B), from a graph G = (V,E), where the fan-out of node i is bounded (d_i), is NP-complete, where* $1 \leq d_i \leq |V| - 2$.

Proof. The problem is in NP, since we can verify in polynomial time, if a candidate spanning tree satisfies the bounded degree constraint as well as the cost of the spanning tree is $< B$.

To show that the problem is NP-hard, we provide a reduction from the Hamiltonian path problem. This proof is not a novel work, but it gives an outline of the NP-completeness proof. The original idea of the proof can be found in [17] [23].

Consider a graph $G = (V,E)$, where V is the set of nodes in the graph and E is the set of links connecting the nodes. Suppose that all the data-products in global set of K data-products are sent on each of the links. Let the cost of each edge $e \in E$ be equal to the latency of data sent on that edge e. Let the cost of the spanning tree be the average end-to-end latency of data reaching the nodes. We need to build a spanning tree with minimum cost, subject to degree constraints on top of this underlying graph. Let the maximum fan-out of a node i be d_i such that $1 \leq d_i \leq |V| - 2$.

Suppose that the fan-out of each node is one. Now if we build a spanning tree with minimum cost on top of G, then the tree will have a depth $|V| - 1$ *i.e.,* the spanning tree will be a path from the root to the last node that is added to the topology. Since this path is a tree, the path goes through each node in the graph only once. Hence, the tree is a Hamiltonian path in the graph. This tree will be constructed only if a Hamiltonian path exists in the graph G. Finding a Hamiltonian path in G is NP-hard [9]. So finding a minimum spanning tree is also NP-hard.

Now we show that even if the degree of each node is between one and d_i, the construction of spanning tree is NP-hard. Consider a node i in graph G. The maximum fan-out of i is d_i. Add d_i-1 nodes to the graph such that these nodes have links only with i. Similarly add nodes and links to G for each node $j \in V$. Let the new constructed graph be G'. Now suppose we construct a minimum spanning tree on top of G' subject to the fan-out constraints of the nodes of G'. This spanning tree will again contain the path (from the root to the last node in G), such that the path goes through each node of G only once. In other words, the spanning tree in G' can be constructed only if a Hamiltonian path exists in G. Since finding a Hamiltonian path in G is NP-hard [9], finding a minimum spanning tree is also NP-hard.

Hence, we can conclude that the problem of finding a minimum cost, degree bounded spanning tree is NP-complete. □

C. Related Work

A problem (Degree Constrained Spanning Trees) of constructing spanning trees with a bound on the fan-out of the nodes, is known to be a NP-hard problem [9]. Another similar problem, of minimizing the cost of a fan-out constrained multicast tree with maximum delay d in a network, is presented in [17]. This problem is also NP-complete

and the idea to prove this fact is given by N. Malouch, Z. Liu and others in [17]. S. Shi and J. Turner in [23] also give a brief idea and methodology to prove this fact.

Once a problem has been proved to be NP-hard, a natural intention is to find an approximate solution that can be computed in polynomial time. Such solutions are suboptimal and are generally designed using a heuristic. Many heuristic algorithms have been proposed in the past to solve the problem of degree-constrained minimum-cost spanning trees.

Shi and Turner in [23] give two formulations of this problem: the first one aims at creating a degree-limited spanning tree with minimum diameter, while the other solution aims at distributing the out-bandwidth usage among the nodes in the topology, while respecting the maximum out-bandwidth capacity. In other words, the first solution attempts to create a flattest possible tree, while the second solution attempts to create a good height-and-width balance in the constructed spanning tree. They also give a proof outline to show that both these problems are NP-complete. The authors propose two heuristic algorithms to solve these problems. The first one is a compact tree algorithm, which is similar to Prim's algorithm for minimum spanning tree and builds the tree incrementally. The second algorithm is a balanced compact tree algorithm, which is a variant of the compact tree algorithm and is used to solve the second formulation of this problem. The performance results of these algorithms show that the balanced compact tree algorithm (which seeks to maintain a height-width balance) performs better than the compact tree algorithm (which is a comparatively greedy approach). These performance results are calculated with respect to the tree diameter, and the rejection rate for new members to be joined to the topology. We will see in Chapter VIII that this fact is true and it applies to our algorithms also. The algorithms given by Shi and Turner construct degree-constrained spanning trees, but their primary focus is to optimize the interface bandwidth at

the multicast nodes. However, our application requires optimization of end-to-end latency subject to constraints on bounded degree of nodes.

Ravi and others in [22] also give two types of problems in network design. These are constructing a Steiner tree and a spanning tree. They show that constructing such trees with bounded-degree and minimum cost is NP-hard. They propose a polynomial-time approximation algorithm to construct a non-uniform but bounded degree, minimum cost, spanning tree. However, the algorithm is restricted to edge-weighted graphs that satisfy triangle inequality. But the routers in real world do not follow the triangle inequality property. So the polynomial time algorithm proposed by Ravi *et al.*, cannot be used for our application.

Malouch *et al.,* in [17] propose a heuristic algorithm to minimize the tree depth of a fan-out constrained multicast spanning tree. The algorithm given in the paper, attempts to create a flat tree, so as to achieve low latency of data delivery. The algorithm constructs tree such that nodes with higher fan-out are closer to the root of the tree. The algorithm depends on the complete knowledge of the underlying topology while constructing the spanning tree. Moreover, the algorithm assumes that the edge-weights of all the edges of the topology are uniform. This constraint makes it difficult to apply this algorithm to our application. However, in case of non-uniform edge-weights, the authors propose a heuristic which includes a variable whose value decides the structure of the topology. They have also provided simulation results to show the nature of the topologies at different values of heuristic.

Su-Wei Tan *et al.*, in [24] propose a new algorithm called Meshtree to construct delay optimized overlay trees. The basic idea in this algorithm is to create a Mesh (as the name suggests) and then construct a spanning tree for data delivery using a path-vector routing protocol. In this algorithm, whenever a new node wants to be added to the topology, a central authority gives a list of potential parents to the

new node. This makes the algorithm dependent on the updated complete knowledge of the topology. Moreover, the performance of the Meshtree depends on the overlay improvement which depends a lot on rearrangement of links. Rearrangement of links can be useful in some applications, but not in this application. One main reason is that rearrangement of links may cause data duplication or data loss which might affect the reliability of data with respect to data completeness. Moreover, rearrangements cause lot of control overhead in execution of the algorithms.

D.Helder and S.Jamin in [11] propose another protocol called switch-trees for creating efficient end-host trees for content distribution. In this protocol, a tree is built initially, and then the nodes in the tree switch their parents so as to construct an efficient tree with minimal latency of data delivery. The performance of this protocol also is dependent on the rearrangement of links. The rearrangements cause additional control overhead, compromise the data completeness, and shift in network traffic. Moreover, the paper did not give any idea about the fault tolerance of the algorithm.

Z.Wang and J.Crowcroft in [35] give outline for choosing a proper heuristic function, while constructing trees that aim to optimize bandwidth and delay at the same time. In this paper, the authors propose two algorithms: one in which the bandwidth is given more priority in the construction of the topology, and another in which latency is given more priority in the decision of the topology. However, no algorithm is discussed, where both the factors can have same importance. Also, no simulation or implementations details (for performance evaluation) are discussed in the paper.

In the next chapter, we will discuss the algorithms used in our solution for the construction of distribution network.

CHAPTER VI

THE PROPOSED SOLUTION

A. Our Approach

We construct overlay trees and combine them to form a Directed Acyclic Graph (DAG) structure and use it as our data distribution network. These trees serve the purpose of creating multiple routes from the unique, common root to the nodes in the DAG structure. The data distribution network is a mesh-based overlay. A spanning tree (a subset of the mesh) is used as the primary means of data transfer. The other trees of the DAG are used for robustness, and hence they are called *backup trees*. Data is also transferred on the backup trees of the DAG, but at slower rate. This creates temporal data redundancy without consuming too much out-bandwidth. The nodes on the backup trees (which send data to the children at a slower rate) are called *backup parents* with respect to the children nodes. The children are called *backed children* with respect to their backup parents.

We use a heuristic approach to construct the topology for data distribution. A heuristic function which is used in the algorithms, is a factor in deciding the structure of the topology. We need to design a good heuristic function so as to construct an efficient topology for data distribution. The design of the heuristic function is done on an intuitive basis as a part of our scientific conjecture. We will perform simulations to evaluate the performance of our designed heuristic and our algorithms. The correctness of our algorithms does not depend on the heuristic function. So, the advantage is that if we find a better heuristic function in the future, we can always use that heuristic function in our algorithms. The algorithms and their correctness will not be affected by this change.

B. Relevant Terms

In this section, we discuss relevant terms which will be used to create the heuristic function. For convenience, consider a node P, its primary parent node R, and a child node Q as shown in the Figure 6. Let N be a new node that wishes to join the topology. The sets with each node represent the data-products required by that node. R requires the data-products {0,1,2,3,4,5}, and being a parent of P, it sends the data-products {0,1,3,5} to P.

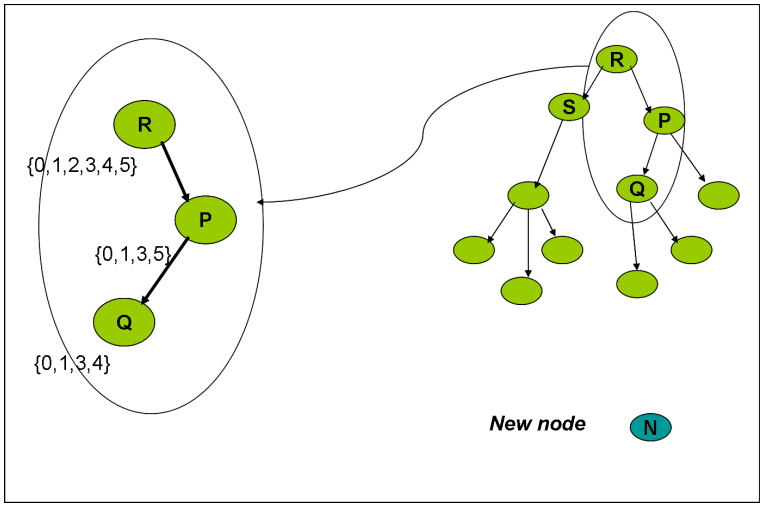

Fig. 6. Tree Construction Method

Saturated. Out-bandwidth of a node P is the resource of a node which is used to satisfy the data requirements of its children. A node P is saturated with respect to the needs of a new node N, if the remaining out-bandwidth with P is not sufficient to send all the data-products in the global set of K data-products to the new node N.

Path. A path from a node X to another node Y is the sequence of links that connect

these two nodes. For example, in Figure 6, the path from R to node Q is R-P-Q (links R-P and P-Q).

Data Hops. A hop from a node X to its child node Y is the link connecting this parent-child pair. A data hop from a parent node X to its child node Y is the data-product that is sent from X to Y. For example, in Figure 6, data hops from R to P are 0, 1, 3 and 5.

Spurious Data Hops. We construct a distribution network that can tolerate at least one crash fault. For achieving this fault tolerance, a node needs to create data redundancy by sending a particular data-product to at least two children. This data redundancy is necessary for tolerating one crash fault. However, if a node X sends a data-product to more than two children, then that data-product is called a spurious data hop. In Figure 6, node R sends data-product 0 to node P. If it sends data-product 0 to two more children, then one of the three data hops is considered a spurious data hop. These data hops consume out-bandwidth of the sending node and hence are not recommended for efficient data distribution.

Superfluous Data Hops. Superfluous data hops are the data-products that a node P needs to request from its parent node R and send to its child node Q just for the fulfillment of the data requirements of its child node Q. In Figure 6, the node Q requires data products 0,1,3,4, but its parent node P does not have some of those data-products (data-product 4). P needs to request that data-product from its parent node R and this request may propagate upwards towards the root of the tree. These data hops (for example, data-product 4) are called superfluous data hops. After these requests are satisfied, the data-products can be sent downstream to the requesting child. Superfluous data hops consume the

out-bandwidth of the node. Also, these data hops create control overhead by sending request messages upwards towards the root of the topology. So reducing the superfluous data hops yields a better tree.

Round Trip Time. The round trip time (RTT) from a node P to a node Q is the time taken by a packet to travel from P to Q and back to P over a network. It is denoted by RTT(P,Q).

We will now define metrics which are used for adding a new node N to the overlay distribution network. These metrics are based on the data-products requirements and the out-bandwidth characteristics of the nodes. The metrics try minimize the out-bandwidth usage, and the end-to-end latency of data at the nodes. The values of these metrics for a node P with respect to N decide the fitness of N being added as a child of P. These values are calculated on the basis of the local state of the overlay, if N is added as a child of P.

Spurious Metric. Spurious metric (r) for a node P with respect to a new node N, is the ratio of the size of all the spurious data hops P will need to send to N to the size of all the data-products required by P. Since spurious data hops consume the out-bandwidth of P, a high spurious metric, r(P,N) reduces the chances of N being added as a child of P.

$$r = \frac{SizeofSpuriousDataHops(node, newNode)}{SizeofAllDataProducts(node)} \tag{6.1}$$

Superfluous Metric. Superfluous metric (s) for a node P with respect to a new node N, is the ratio of the size of all the superfluous data hops P will need to send to N, to the size of all the data-products required by P[1]. Like spurious

[1]Data-products required by P means all the data-products that P uses as well as

metric, a high superfluous metric of P also reduces the chances of N being added as a child of P.

$$s = \frac{SizeofSuperfluousDataHops(node, newNode)}{SizeofAllDataProducts(node)} \tag{6.2}$$

RTT Metric: The RTT metric (d) of a node P with respect to a child node N, calculated by P's parent node R, is the ratio of the RTT(R,N), to the sum of RTT(P,N) and RTT(R, P).

R makes a decision whether to add the new node N as its child or to throw it to node P for addition. R can add N as a child and send data-products to N. However, R needs to estimate the latency by which data-products will reach N if R sends them through its child P. This estimation is done using the RTT metric.

$$d = \frac{RTT(node, newNode)}{RTT(node, childNode) + RTT(childnode, newNode)} \tag{6.3}$$

The higher the RTT metric, the lower is the latency. Moreover, if the RTT metric of child P is greater than 1, the data-products from R to N will reach faster if they are routed through P rather than sent directly to N. This formula helps R to find out which one of its children can send the data-products to the new node N with a lower latency. So, whenever R decides to throw N to one of its children, RTT metric will increase the chances of N being thrown to children nodes having higher RTT metric.

Products Metric: Products metric (p) of a node P with respect to a child node N, is the ratio of the size of number of the data products that P can satisfy for N,

---

the data-products that P does not use, but needs to send downstream to fulfill the requirements of its descendants.

to the size of all the data products required by node N.

$$p = \frac{SizeofAllDataProductsMatched(node, newNode)}{SizeofAllDataProducts(newNode)} \tag{6.4}$$

A high $p(P,N)$ for a node P with respect to new node N always increases the chances of N being added as a child of P.

Out-bandwidth Metric: Out-bandwidth is the bandwidth consumed by the data flowing out of that node. This resource bounds the number of data hops a node can send downstream and hence fan-out of a node. The out-bandwidth metric o of a node N is the ratio of the out-bandwidth remaining with N to the maximum out-bandwidth (*i.e.,* the resource capacity of the node) of N.

$$o = \frac{outBandwidthRemaining(node)}{outBandwidth(node)} \tag{6.5}$$

A high value of out-bandwidth metric of a node P means that very less of its out-bandwidth has been used and so increases the chances of new node being added as a child of P.

Overall Metric: The overall metric for a node is

$$m = d + o + p - r - s \tag{6.6}$$

As mentioned before, the metrics d, o and p increase the chances, and metrics r and s decrease the chances of a new node N being added as a child of P. We have designed this metric on an intuitive basis as a part of our scientific conjecture. There can be many different ways of constructing this heuristic function, but a linear combination is simple to evaluate. This heuristic function may not be the most efficient one, so we need to evaluate the performance of this function through simulations. We will evaluate the performance of this

heuristic in Chapter VIII on the basis of comparison with the existing solution and a randomized solution which is discussed later in Chapter VIII.

C. Algorithms for Topology Construction and Management

The Unidata solution has different distribution networks for each data-product. Since the number of data-products is ever increasing, maintaining so many topologies will be challenging. In our solution, we construct a single network that is common for all data-products. As mentioned before, we construct multiple trees, and combine them to form a DAG structure. A spanning tree which is a subset of the DAG is used for primary data transfer. The paths which are not on the primary tree are called backup paths and they are used for backup (redundant) data transfer. The tree components of DAG are designed in such a manner that there exist multiple routes from root to each node. The multiple routes create data redundancy and help in tolerating crashes. We have written an algorithm to join a new node to the topology. The overlay DAG structure is incrementally constructed by calling this algorithm for each node, when it requests to be a part of the IDD system.

1. Join a Node to the Topology

The algorithm to join a new node to the topology is shown in Figure 7. If a new node N wants to join the topology, it is initially handled by the root R (*i.e.,* the Unidata server). R has to make a decision whether to add the new node N as its own child, or to throw N to one of its children for addition. This decision is made locally (*i.e.,* the decision does not depend on the complete knowledge of the topology). If R is not saturated, it calculates the metrics p, o, d, r and s for itself with respect to the new node N. It then calculates the overall metric for itself and stores it in a List L

(step 3 in the algorithm). R then calculates the metrics p, o, d, r, s and the overall metric for each child C_i of R and stores the overall metric in the List L. It then finds the nodes with maximum overall metric and second maximum overall metric. These nodes are called C_m and C_{sm} in the algorithm (Step 5).

If R is the node with maximum overall metric (*i.e.,* C_m), it adds the node N as its own child using the *add-child* algorithm. The *add-child* algorithm is explained later in the thesis. R may also add N as a child, in case of a *push* operation (step 6.1 in the algorithm). This operation is explained later in the subsection 1.a.

However, if the above cases are not true, R throws N to the child node with best overall metric (*i.e.,* C_m) for addition (Step 6.1.1). If R is the node with the second best overall metric (*i.e.,* C_{sm}), then the node R backs the new node N, or else it throws N to C_{sm} for backup (step 6.1.2 in the algorithm).

For each new node N joined to the topology, it is either added as a child by the root or thrown to two nodes: one for addition (say node P) and another for back up (say node P'). Hence for each node N (except direct children of the root), at least two routes are initiated from the root. Node P then runs the *add* algorithm and node P' runs the *back* algorithm. These algorithms are shown in Figures 8 and 9, respectively.

Algorithm to join a node to the topology

Comments: This algorithm is run by the Unidata server (root node) to join a node to the topology. Whenever a node *n* requests to be joined to the topology, Unidata either adds *n* as its child or throws *n* to a child node for addition. Similarly, Unidata either backs *n* as its child or throws *n* to another child node for backup.

Var: *n* - Node to be added,

L – List to store overall metrics of nodes.

out_bandwidth(i) – Resource capacity of a node *i*.

join_and_back(node *n*)

1. Get the data-product requirements of *n*.
2. **If** saturated with respect to the data-product requirements of *n* go to step 4.
3. Calculate Unidata's overall metric with respect to the new node *n* and add the metric to list *L*.
4. For each child C_i
   1. Calculate its overall metric with respect to new node *n* and add the metric to the list *L*.
5. Find the maximum and second maximum overall metric from list *L* and let the corresponding nodes be C_m and C_{sm} (if it exists).
6. **If** Unidata has the maximum overall metric (it is the node C_m), add the node *n* using **add_child** algorithm.

   else

   1. **If** out_bandwidth(*n*) > out_bandwidth(C_m), push *n* between Unidata and child C_m using **push(*n*)** algorithm

      else

      1. Throw the node *n* to child with max overall metric C_m, for addition.
      2. **If** Unidata is the node with second max metric *i.e.,* C_{sm}, back the node *n* using **back_child** algorithm.

         else throw the node *n* to child C_{sm} for backup.

Fig. 7. Algorithm to Join a New Node to the Topology

Algorithm to add a node to the topology

Comments: This algorithm is run by a node *p* to add a node to the topology. Whenever *p* gets a request from its parent node to add node *n*, *p* runs this algorithm. Using this algorithm, *p* either adds *n* as a child node or throws *n* to one of its children for addition.

Var: *n* - Node to be added,

L – List to store overall metrics of nodes.

out_bandwidth(p) – Resource capacity of a node *p*.

add(node *n*)

1. Get the data-product requirements of *n*.
2. **If** saturated with respect to the data-product requirements of *n* go to step 4.
3. Calculate *p*'s overall metric with respect to the new node *n* and add the metric to list *L*.
4. For each primary child C_i
   1. Calculate its overall metric with respect to new node *n* and add the metric to the list *L*.
5. Find the maximum overall metric from list *L* and let the corresponding node be C_m
6. **If** *p* has the maximum overall metric (it is the node C_m), add the node *n* using **add_child** algorithm.

   else

   1. **If** out_bandwidth(*n*) > out_bandwidth(C_m), push *n* between *p* and child C_m using **push(*n*)** algorithm

      else

      1. throw the node *n* to child with max overall metric C_m, for addition.

Fig. 8. Algorithm to Add a New Node

As shown in the *add* algorithm, P either adds the new node N as its child, or it hands over control to the child having highest overall metric for addition. Similarly, the node P', running the *back* algorithm, either backs the N, and adds it as a backed child, or throws N to the child having highest overall metric for backup. So the control is transferred from R to P and P', and then may be passed from these nodes to their children. Eventually, when N is added to the topology, it gets a parent (P or its descendant) and a backup parent node (P' or its descendant). We can see that node-disjoint routes are created from the root to the new node. Combining all such routes for each node in the topology, we get a DAG structure for data distribution.

Algorithm to back a node

Comments: This algorithm is run by a node *p'* to provide a backup parent for *n*. Whenever *p'* receives a request from its parent node to back node *n,* it runs this algorithm. Using this algorithm, *p'* either backs *n* or throws *n* to one of its children for backup.

Var: *n* - Node to be backed,

L – List to store overall metrics of nodes.

out_bandwidth(p') – Resource capacity of a node *p'*.

back(node *n*)

1. Get the data-product requirements of *n*.
2. **If** saturated with respect to the data-product requirements of *n* go to step 4.
3. Calculate the overall metric of *p'* with respect to the new node *n* and add the metric to list *L*.
4. For each parimary child C_i
    1. Calculate its overall metric with respect to new node *n* and add it to the list *L*.
5. Find the maximum overall metric from list *L* and corresponding nodes be C_m
6. **If** *p'* has the maximum overall metric (it is the node C_m), make the node *n* a backed child using **back_child** algorithm.

    else

    1. Throw the node *n* to child with max overall metric C_m, for backup.

Fig. 9. Algorithm to Back a New Node

Unidata hands over the control to join the new node N to either P, or P' or both and exits the algorithm *join-and-back*. Suppose that the nodes P or P' crash while the algorithm shown in Figure 7 is still running. In that case, the control may not reach either of the nodes, and hence N will not get a parent or a backup parent.

Similarly, N may not get a parent node if P is running the *add* algorithm and one of P's children crashes. N may node get a backup parent if P' is running the *back* algorithm and one of P''s children crashes.

In such a situation, N requests Unidata again for a new parent and a backup parent.

a. Push a New Node between a Parent-Child Pair

For any parent child pair (P,C) the resource capacity of the parent node P is never less than the resource capacity of the child node C. Consider the three nodes R, P and Q as shown in the Figure 11. Suppose a new node N wants to join the topology, and the node making the decision is R. Thus, R has the control to add the node N to the topology. If R finds that P is the node with the best overall metric and also that N has more resource capacity (out-bandwidth) than P has, it pushes the node N between itself and the node P. This push operation is done using the algorithm shown in the Figure 10. According to the algorithm, R adds N as its own child and only when P starts receiving data-products from N, R cuts-off the link R-P. Since, P is temporarily accepting data-products from both R and N, we assume that P has enough in-bandwidth capacity to do so. The resulting topology is shown in the Figure 12.

Algorithm to push a node in the topology

Comments: This algorithm is run by a node *r*. Let *n* be a node that wants to be added to the topology. Let *p* be the child of *r* with maximum overall metric with respect to *n*. Suppose *n* has less resource capacity than the controlling node *r*, but more resources than *p*. In this case, *r* pushes the new node *n* between itself and its child *p* with best overall metric with respect to *n*.

push(*n*)

1. Add the node *n* as a child using **add_child** algorithm.
2. Request the node *n* to add the child *p* with the best overall metric with respect to *n*.
3. Query the Failure detector to find if *n* has crashed. **If** *n* has crashed, prune off the link with *n* and exit the algorithm.
4. **else** when *p* starts receiving data-products it requires from another parent, prune off the parent-child link with *p*.

Fig. 10. Algorithm to Push a Node in the Topology

R queries its failure detector just before R executes this algorithm, to find out crashed nodes. However, consider the case, where the node N crashes while R is still executing this algorithm. According to Step 3 in the algorithm, R detects that N has crashed and cuts-off the link with N. The original link R-P is not pruned. Thus, the distribution topology remains the same as it was.

A sample topology constructed using our algorithms is shown in the Figure 13. The bold paths show the primary tree of data transfer, while the dotted lines show the backup links of data transfer.

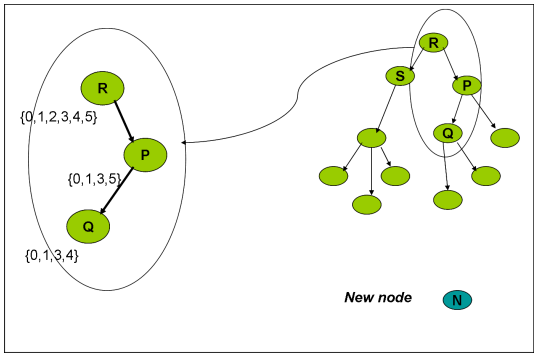

Fig. 11. A Sample Graph to Explain *Push* Operation.

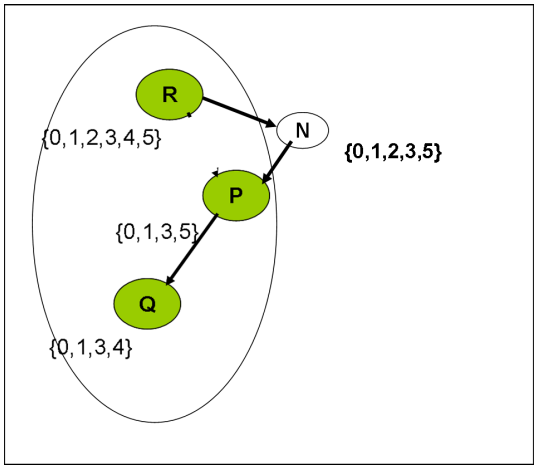

Fig. 12. N Pushed between the Node R and Its Former Child P

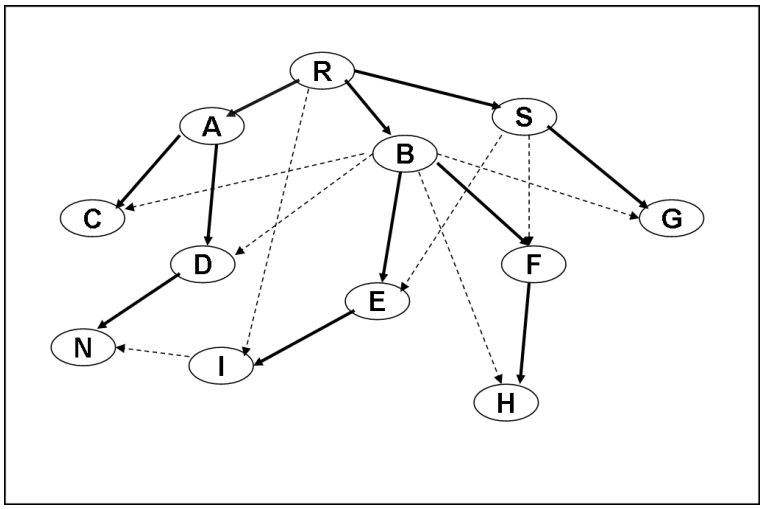

Fig. 13. A Sample Topology Created Using Our Algorithm

2. Add a Child Node

The algorithm run by a node P to add a node N as a child is shown in the Figure 14. If P (which has the control to add the new node N) finds that the overall metric of P with respect to addition of N is better than each of its children's overall metric, then P executes this algorithm to add N as its own child. This algorithm adds N to P's children list. It reduces the *out-bandwidth-remaining* of P by a value equal to the total size of the global set of K data-products[2]. Note that the node P reserves the bandwidth equal to the total size of the global set of K data-products for the newly added child node N, even if the data-requirements of N are less than the total size of the global set of K data-products. This is done because in the future, N may request extra data-products for superfluous hops. Some data-products that N requires, but are not present with P (superfluous hops) are requested by P to its

[2]A node n uses its resources to send data-products to its children and backed children. The out-bandwidth-remaining of a node n is a term for the remaining resources with a node n.

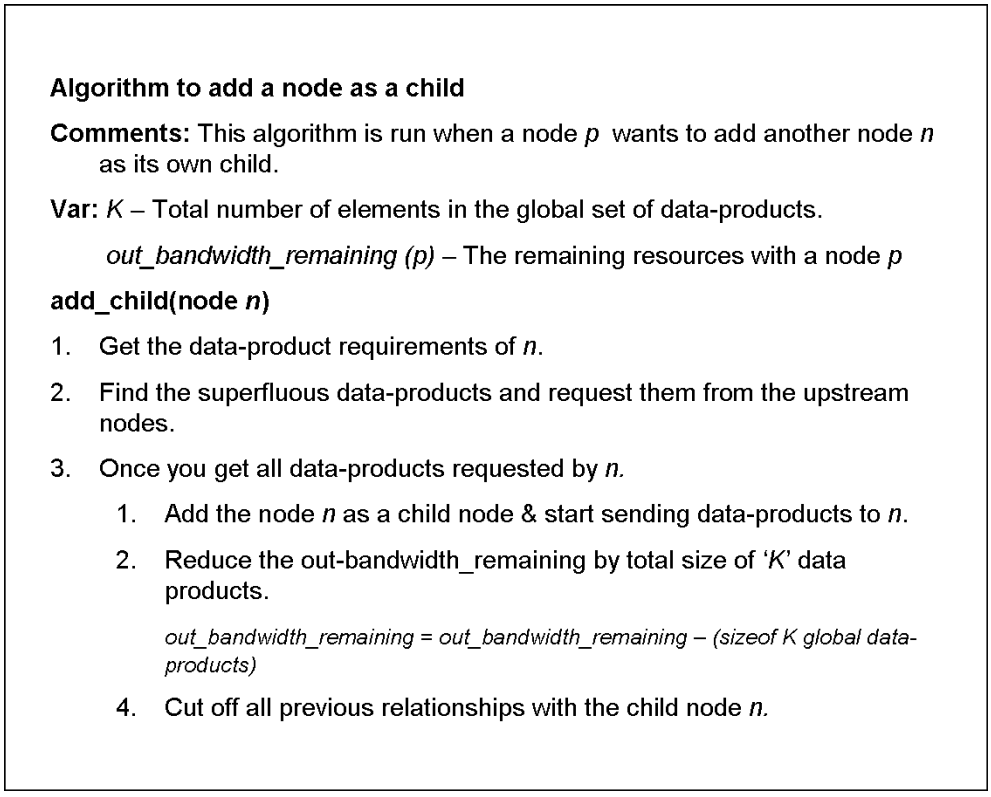

Fig. 14. Algorithm to Add a New Node as a Child

parent node. These requests propagate upwards towards the root, until P receives all the requested data-products. Once P receives all the data-products that N requires, P starts sending those data-products to N.

We can see from Step 3 in the algorithm, that only after P receives all the data-products that N requires, it starts sending those data-products to N. P queries its failure detector just before executing this algorithm to find out crashed nodes. However, consider the case where a node in the primary path of P crashes while P is still running this algorithm. In such a case, lot of rearrangements take place, and the topology takes a finite time to stabilize. So the requested data-products (superfluous hops) can take a long time before they reach P and hence node N.

Algorithm to back a node and add it as a backed child

Comments: This algorithm is run when a node *p'* wants to back a node *n* and add it as its own backed child.

Var: *K* – Total number of elements in the global set of data-products.

out_bandwidth_remaining (p') – The remaining resources with a node *p'*

back_child(node *n*)

1. Get the data-product requirements of *n*.
2. Find the superfluous data-products and request them from the upstream nodes.
3. Once you get all data-products requested by *n.*
    1. Back the node *n* as a backed child node & start sending data-products to *n*.
    2. Reduce the out-bandwidth_remaining by total size of '*K*' data products.

       out_bandwidth_remaining = out_bandwidth_remaining – (sizeof K global data-products) /5
    4. Cut off all previous relationships with the backed child node *n.*

Fig. 15. Algorithm to Back a New Node and Add as a Backed Child

3. Backup a Node

As described in the Section 1, if R does not add the new node N, then it may back the new node acting as its backup parent, or it may throw N to one of its children (say P') for backup. Note that if R adds N as its own child, it does not provide for any backup. This is because direct children of the root do not require a backup parent. If the root crashes, then obviously no one receives data and can back N.

Each node in the topology stores at least one hour's worth of data in its local product-queue to send the data-products to its children and backed children. A primary parent sends this data to the children nodes at a faster rate all at a time.

However, a backup parent sends these data-products to its backed children at a slower rate. The data-products are not sent all at the same time, but at certain time intervals within one hour. As of July 2006, we noticed that the total size of data-products sent downstream and the product-queue size[3] are such that, the data-products can be sent in five slots within one hour to the downstream children. So, we designed our algorithms such that the backup parents should reserve only as much bandwidth (to send slow data-products to backed children) as is required. In the future, if the data-products size or the product queue size increases, this factor will change.

Consider that *P'* decides to back *N* and add *N* to its backed children list. *P'* runs the back child algorithm shown in Figure 15 to do this. The out-bandwidth of *P'* is reduced by one-fifth portion of the total size of global set of k set of data-products, since *P'* will be using only that much bandwidth to send the slow backup data hops to *N*. *P'* sends messages to its parent node requesting the data-products that *N* requires but are not present with *P'* (superfluous hops). Once *P'* receives all the data-products required by *N*, *P'* starts sending the data-products to *N* at a slower rate. If *N* has already received the data-products from its primary parent at a faster rate, *N* sends a notification to *P'* about it. Consequently, *P'* does not send that batch of data-products to *N* again.

We can see from Step 3 that only after *P'* receives all the data-products that *N* requires, it starts sending those data-products to *N*. *P'* queries its failure detector just before the algorithm to find out crashed nodes. However, consider the case where a node in the primary path of *P'* crashes while *P'* is still running this algorithm. In such a case, lot of rearrangements take place, and the topology takes a finite time to stabilize. So the requested data-products (superfluous hops) can take long time

[3]The actual values of data-product sizes and product queue size can be obtained from [28].

before they reach *P'* and hence node *N*.

4. Failure Recovery

In this section we present algorithms for failure recovery. We present algorithms for both the types of node pairs. (1) parent-child pair and (2) backup parent-backed child node pair.

a. Failure Recovery for Parent-Child Pair

Consider a parent node P and its child node C. We will now discuss how faults are handled. The parent node P sends data-products to the child node C. Hence P is called the *sending node*, while C is called the *receiving node*.

The sending node P performs a query to its failure detector before it sends the data-products to the child node C. If the failure detector at the node P suspects C, then C has crashed (by the strong accuracy property of perfect failure detector). So the sending application running at the node P stops sending data-products to the child node.

Also, the child node C queries its failure detector to determine whether any of its neighbors have crashed. If the failure detector of the child node suspects that the parent node P has crashed, then C finds another parent node. It does this using the algorithm shown in the Figure 16. Initially it queries its backup parent *P'* for being a new primary parent. If *P'* has enough resources and becomes the parent, then the backup connection is changed to a primary connection. Consequently, C loses a backup parent and then looks for a new backup parent using the algorithm *find-new-backup-parent* as shown in the Figure 17. This algorithm is discussed later in Subsection C.b. However, if its backup parent does not have enough resources to become a new parent for C, it (the node C) requests the root for another parent.

Algorithm to find a new parent when original parent node crashes

Comments: When the parent of a node *c* crashes, the node *c* has to look for another parent node. It initially requests its backup parent node to be new primary parent. If the backup parent does not have enough resources to become the new primary parent for *c*, then *c* requests the root to get a new parent and new backup parent. After this operation, *c* gets a new primary as well as a new backup parent, and hence the primary paths of all the nodes in the subtree of *c* change accordingly.

find_new_parent(node c)

1. Request the backup parent to be new parent node.
2. **If** the backup parent agrees to be a new parent,
    1. Change the corresponding connection from backup connection to primary connection.
    2. find_new_backup_parent(c)
3. **else**
    1. Request the root for a new parent and new backup parent using the **join and back** algorithm.
    2. Cut the connection with previous backup parent.

Fig. 16. Algorithm to Find New Parent

In that case, C requests to be added to the topology as if it is a new node, using the *join-and-back* algorithm. C gets another parent and backup parent nodes. It then cuts-off the relationship with the previous backup parent P'. Since C gets a new primary and a new backup parent, the primary path of each node in the subtree of C changes accordingly. However, we will prove in next chapter that even if this rearrangement occurs, all the nodes in the topology will have node-disjointness.

b. Failure Recovery for Backup Parent and Backed Child Pair

The algorithm at the backup parent (say P') is almost the same as the algorithm for a parent node. If its failure detector suspects that the backed child node C has crashed, P' stops sending data-products to the backed child node C.

Algorithm to find a new backup parent when a backup parent node crashes or a node loses a backup parent

Comments: When the backup parent of a node *c* crashes, *c* looks for another backup parent. It sends a request to its parent for new backup. The parent in turn forwards the request upstream. This algorithm is run by a node *i*, when it receives a request to find another backup parent for *c*. The request travels upwards on the primary route until it reaches a node which is not a backup parent of *n* and also has the capacity to back *c*.

As a result of a push operation, a node may realize that it is not a direct child of the root node anymore, and also it does not have a backup parent. In such a situation, that node executes this algorithm and requests for another backup parent.

find_new_backup_parent(node c)

1. Receive request for find_new_backup_parent for node *n* from downstream nodes.
2. **If** *i* is not (parent or backup parent of *c*)
    1. If *i* has enough out-bandwidth remaining to back *c*, add it as a backed child.
3. **else** send the find_new_backup_parent request upwards.

Fig. 17. Algorithm to Find New Backup Parent

On the other hand, if the failure detector of the backed child C suspects that the backup parent node P' has crashed, then it requests for another backup parent using the algorithm shown in the Figure 17. The node C sends the request to its parent node, which in turn sends the request to its parent node for finding a new backup parent for C. Any node (which receives the request of finding new backup parent) who is not the neighbor of C and also has resources to be the backup parent of C, becomes the new backup parent of C. The process of finding a new backup parent is shown by the algorithm in the Figure 17. We will prove in the next chapter that after C gets another backup parent, and it also satisfies node-disjointness.

The node C gets a new backup parent after the execution of this algorithm. The

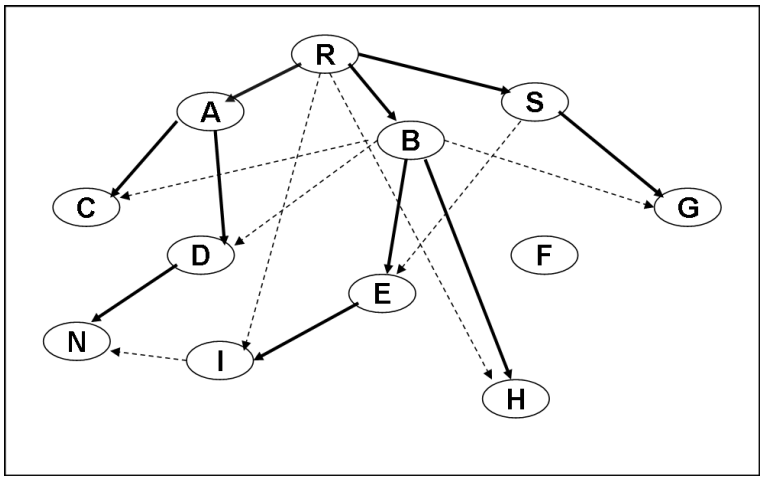

Fig. 18. An Existing Node F is Removed from the Topology

find-new-backup-parent request is sent by C and it travels upwards towards the root. Suppose that one node on the primary path of C crashes. Due to this, the *find-new-backup-parent* request may be lost, and C may not get a new backup parent. Such a situation occurs, when a node crashes and before the topology is stabilized another crash occurs. This shows that our algorithm can not tolerate more than one crash at a time and also requires a finite time interval for the topology converges to a stable state.

If a node is crashed, then due to the strong accuracy property of failure detector P, its parent and backup parent nodes detect that the node has crashed. Consequently, they stop sending data-products to the node. Similarly, its children nodes and backed children nodes detect that the node has crashed and find new parent and backup parent nodes respectively. In other words, the node is removed from the topology (it loses connections with all its neighbors). Figure 18 shows the change in the topology when a node F is crashed and it is removed from the topology given in the Figure 13. Since the node F is crashed, its parent and the backup parent nodes

stop sending data to F. The child node H does not receive data-products from F. Also, its failure detector suspects F has crashed. So, it requests its backup parent B to be its new parent node. When B becomes a new primary parent of H, it (the node H) loses a backup parent. Consequently, it requests for another backup parent using the algorithm shown in Figure 17. As a result, R becomes the new backup parent for H and a new backup link R-H is added to the topology. The rest of the links in the topology remain intact.

In this chapter, we presented algorithms to construct the topology for data distribution. We also presented algorithms to tolerate faults and make appropriate changes to the topology. However to ensure that these algorithms are correct and do not lead the topology in such a state that cannot be handled, we will discuss the analysis of the algorithms and lemmas to prove the correctness of the algorithm in the next chapter.

CHAPTER VII

ANALYSIS OF THE ALGORITHMS

In this chapter we discuss the proof of correctness of the algorithms presented in Chapter VI. The safety, progress, and tolerance properties of the problem are discussed in Chapter IV Section B. We will prove these properties in this chapter.

A. Proof of Correctness

Before we can start with the proofs, we give definitions for some relevant terms.

Consider Figure 19. The primary tree is showed by bold lines and the backup links are shown by dotted lines.

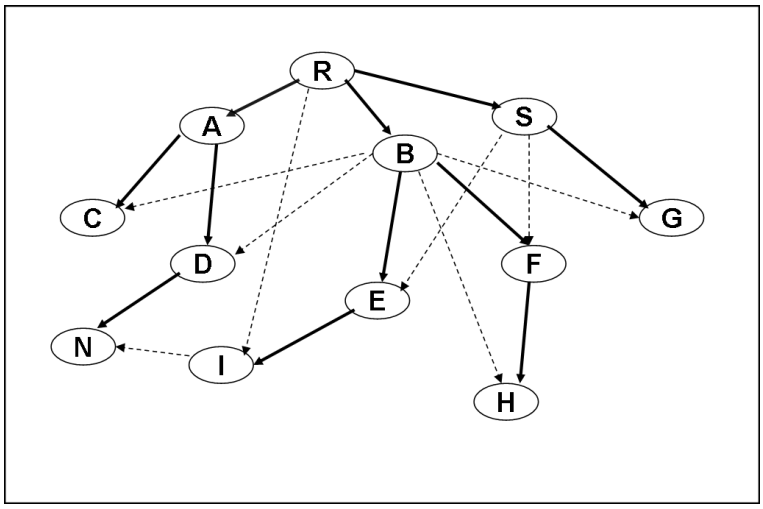

Fig. 19. A Sample Topology Created Using Our Algorithm

Definition A.1. *Primary route of a node N is defined as the sequence of links on the primary tree from the root to N.*

The primary route of the node A in Figure 19 is R-A, while the primary route of H is R-B-F-H.

Definition A.2. *The depth of a node in the primary tree is defined as the number of links in the primary route of the node from the root of the tree.*

The depth of the nodes R, A, D, and H in Figure 19 are 0, 1, 2, and 3 respectively.

Definition A.3. *An intermediate node I on primary route of a node N is a node on the primary path of N except the root and N.*

From Figure 19, D and A are intermediate nodes on the primary route of N. S is an intermediate node on the primary route of G.

Definition A.4. *A parent subtree of a node N is a subtree of the primary tree whose root is primary parent of N.*

The parent subtree of E is the subtree of the primary tree rooted at B *i.e.*, the subtree having the nodes B, E, F, and H.

Definition A.5. *Node-disjointness property for a node N is defined as "For each intermediate node I on the primary route to N, there exists at least one route $Route_I$ from the root to N such that $Route_I$ does not contain I".*

This property means if any one node I on the primary route to a node N crashes, the data can be delivered to the node N through an alternate route $Route_I$ which bypasses the crashed node I. For example, in Figure 19, if node A crashes, data can be sent to N through route R-B-D-N. If D crashes, data can be sent through route R-B-E-I-N. Thus, for each intermediate node (A or D) on the primary path of N, there is at least one route from the root to N (R-B-D-N or R-B-E-I-N respectively) such that the route does not contain the intermediate node. Hence, N has node-disjointness.

It follows from the definition that direct children of the root do not have any intermediate nodes in their primary path. Hence, they have node-disjointness by default.

In the following part of this section we discuss lemmas to prove the correctness of our algorithms.

Lemma A.1. *Consider a node N, its parent node P and its backup parent node B. If*

1. *the primary route (say route1) from root to P and*
2. *at least one route (say route2) from root to B*

do not have any common nodes (except the root), then the node N has node-disjointness property.

Proof. Consider two routes from the root to N: one through P and the other through B. Let the route through P include the links on *route1* appended by the link P-N, and let the route through B include the links on *route2* appended by the link B-N. Since *route1* and *route2* do not have any nodes in common (except the root), the route from root to N through B does not include any of the intermediate nodes on the primary route of N. Hence, by Definition A.5 we conclude that N has node-disjointness. □

Lemma A.2. *Consider a node N, its parent node P and its backup parent node B. If P has node-disjointness and B is not in the parent subtree of N, then node N has node-disjointness.*

Proof. If the node P has node-disjointness, there are routes from the root to P, such that for each intermediate node I on the primary route to P, there exists some route from the root to P that does not contain I. In other words, for each intermediate node I, we can find some route from the root to P that bypasses the node I.

Consider the routes to N that are constructed by appending the link P-N to all the routes to P. It follows that for each intermediate node (except P) on the primary route to N, we can find some route of these constructed routes that bypasses the intermediate node.

Also consider a route from the root to N that is constructed by appending the link B-N to the primary path of B. Since B is not in the parent subtree of N, it is not a descendant of P and hence does not have P in its primary path. The primary path to B appended by link B-N can thus bypass the node P.

Thus we have constructed routes from root to N. These routes are such that for each intermediate node I in the primary path of N, we can find at least one constructed route that does not contain I. Hence, by Definition A.5 we conclude that N has node-disjointness. □

Lemma A.3. *If for a node N, both its parent node P and its backup parent node B have node-disjointness, then node N also has the node-disjointness property.*

Proof. Since the node P has node-disjointness, there are routes from the root to P, such that for each intermediate node I on the primary route to P, there exists some route from root to P that does not contain I.

Let us append all the routes to P with the link P-N, to construct routes to N. It follows that for each intermediate node (except P) on the primary route to N, we can find some route from these constructed routes that bypasses the intermediate node.

Now we need one route from the root to N such that the intermediate node P is not present on that route. This route is available by appending the routes from the root to backup parent B with the link B-N. For the position of B there are two possibilities.

- B is not in the parent subtree of N. This means B is not a descendant of P and

hence does not have P in its primary path. It follows that one route from the root to N (primary path of B appended by link B-N) does not contain P.

- B is in the parent subtree of N. In this case, P lies on the primary path of B. However, B has node-disjointness. So there exist at least one route from the root to B (and hence to N) such that it does not contain P.

As a result, we find that if both the parent as well as backup parent nodes of N have node-disjointness, then for each intermediate node in the primary route to N, there exists some route to N that bypasses the intermediate node. Hence from Definition A.5 we conclude that the node N also has node-disjointness. □

Consider a node N, its parent node P and its backup parent B. For the backup parent of N there are two possibilities.

1. The backup parent is not in the parent subtree of N. In this case, the node-disjointness property of N depends on the node-disjointness property of its primary parent node (Lemma A.2).

2. The backup parent of N is in the parent subtree of N. In this case, the node-disjointness property of N depends on the node-disjointness of P as well as B. (Lemma A.3)

We define a term *position* as follows:

Definition A.6. *The position of a node N in the topology is defined as*

- *1 (if the node is a direct child of the root),*
- *The position of its primary parent node (say P) + 1 (if its backup parent (say B) is not in parent subtree of N), or*

- *The position of its backup parent node B + 1 (if B is in the parent subtree of N).*

Lemma A.4. *The position of a node N is always greater than the position of its primary parent node.*

Proof. Before we start the proof, let us define a *topological sort* of a DAG structure $G = (V,E)$, where V is set of vertices and E is set of edges of the graph. Topological sort is an operation done on a DAG structure, so as to obtain a linear ordering of the vertices V of graph. This linear ordering is such that for every directed edge $(i,j) \in E$, vertex i is to left of j.

Consider a node N, its primary parent P and its backup parent B. The topology created by our algorithms is a DAG. In this topology the primary tree is shown by bold paths, while the backup links are shown by dotted paths. We perform little modification to the DAG structure. If a node has its backup parent in its parent subtree, then the link between the node and its backup parent is made bold. Now, we still have a DAG structure, and from this structure we remove all the dotted links. We perform a topological sort on the remaining bold lines of the DAG. After the topological sort is done, we get a linear ordering of the nodes of the graph. Let us give some weights to the nodes in the linear ordering. Let the root have the value 0, and every node that is x hops from the root have the value x. In case, a node ends up with two or more values (because of two or more paths from the root), the largest value is given to the node.

We see that the value of each node in this linear ordering is the position of each node in the original DAG topology. From this linear ordering, it is clear that for each node N, its parent node P has fewer hops from the root than node N. Hence, the value of a node N is $>$ the value of P. Hence, for each node N in the DAG, the

position of $N >$ the position of its parent P. □

1. Safety

Lemma A.5. *Whenever a new node is added to the topology, the node-disjointness of each node in the topology is unaffected.*

Proof. According to the *join-and-back* algorithm shown in the Figure 7, a new node can be added to the topology as a leaf node or as an intermediate node. We prove that in both the cases, the node-disjointness of each node in the topology is not affected. In Part 1 of proof, we consider the case that N is added as a leaf node. This happens when the control to add the new node N is transferred from the root to one of its children, then to one of its grandchildren, and so on until the control reaches a leaf node. Finally the leaf node, adds N as its child. We prove that in this case, the node N has node-disjointness and the node-disjointness of other nodes in the topology is unaffected. In Part 2 of the proof we prove that even if the new node N is added to the topology as a non-leaf node (using the *push* algorithm shown in Figure 10), the node-disjointness of each node in the topology is not affected.

Part 1: The addition of a node N to the topology does not cause any re-arrangement of links. New links are added to the topology. These links are the links between the node N and its parent and the backup parent nodes. We will prove that in this case, the new node has the node-disjointness property, and also that the node-disjointness of other nodes in the topology is unaffected.

Let R be the root of the topology and let R_i and R_j be any two distinct children of R. Let the children of R_i be R_{ik} where k= 1 to m (for example R_{i1}, R_{i2}, R_{i3} ..) and let R_{jl} where l= 1 to m' be the children of R_j. We follow such a nomenclature for each parent-child pair in the topology. Thus, each intermediate node under the

primary subtree of R_i has a prefix $R_{i...}$ in its name. Similarly, any node under the primary subtree of R_j has a prefix $R_{j...}$ in its name.

If R adds the node N as its child, then N is a direct child of the root and by Definition A.5, N has node-disjointness. Also, if R backs the node N, the route R-N can bypass each intermediate node in the primary path of N and hence N has node-disjointness.

Suppose that R neither adds N as a child nor backs N as a backed child. Without loss of generality suppose that the new node N is thrown by the root to R_i for addition and to R_j for backup. R_i in turn may add the node as its own child or throw it to one of its children for addition. In either situation, the new node is added under the primary subtree of R_i. Similarly, N is backed by any node under the primary subtree of R_j. Let the parent of N be P, and the backup parent of N be B. Thus, for the primary path from the root to N, every intermediate node has a prefix $R_{i...}$ in its name and for the backup path from the root to N, every intermediate node has a prefix $R_{j...}$ in its name. Because of our assumption that R_i and R_j are two distinct nodes, the subtrees of the nodes R_i and R_j are also disjoint. Hence, we can conclude that both the primary as well as backup path to the new node N do not have any common nodes (except the root). Thus, we can conclude (from Lemma A.1) that the node N satisfies node-disjointness. It is also evident from the *add* and the *back* algorithms that in this case, no other links in the topology are cut off or rearranged. Hence, the parent and the backup parent of each node in the topology remain the same. Consequently, the node-disjointness of each node in the topology is unaffected.

Part 2: The new node N can be added as a non-leaf node using the *push* algorithm. In this case, we show that N has node-disjointness. We also show that the addition of N does not cause any other node to lose node-disjointness.

Suppose a parent node P adds the new node N as its own child using the *push*

algorithm. As such, the original child C of P now becomes a child of N. When N is being added to the topology a backup parent is assigned to it (except the case where P is the root of the topology, in which case N does not need a backup parent). Using the same argument that we used in Part 1, we can show that both the primary paths of the parent as well as backup parent of N do not have any intermediate nodes in common. Hence, (from Lemma A.1) N has node-disjointness.

According to the *push* algorithm the P-C link is cut-off. A new link N-C is added to the topology. It is evident that no other links of the topology are rearranged or cut-off. Hence, the nodes which can lose node-disjointness due to the addition of N are the node C and the nodes in the primary subtree rooted at C. If C initially had no backup parents (P is the root), then C asks for another backup parent using the algorithm shown in Figure 17. If C initially had backup parent, it is evident from the *push* algorithm that the backup parent is not in the subtree of the new node N (parent subtree of C). Also, we have shown in Part 1 that N has node-disjointness. Hence, by Lemma A.2 the node C also has node-disjointness. Consequently, all other nodes in the primary subtree of C have node-disjointness.

Thus, from Part 1 and Part 2, we conclude that after a new node is added to the topology, all the nodes in the topology have node-disjointness. □

2. Progress

Lemma A.6. *If a new node is added to the topology, eventually after a finite time interval, the new node starts receiving all the data-products it requires. This stability will occur with the assumption that no faults occur in the system during the finite time interval.*

Proof. When a new node (say N) is added to the topology, one parent node (say P) is

chosen to supply data-products to N. The parent P receives requests from N regarding the data-products that N requires. As discussed in Chapter VI in SubsectionC.2, if P does not have some data-products that N requires, it sends a request (control message) to its own parent (say P_1) for these data-products. P_1 may in turn forward the control message to its parent, and such requests may propagate up towards the root of the tree until they are satisfied. If the parent node, who receives the request, has all of the requested data-products it starts sending them downstream. We prove that after a finite time interval, no more control messages are sent, and the new node N receives all the data-products it requires. However, if a new node N is added by the *push* algorithm shown in Figure 10, rearrangement of links takes place. In this case, we show that data-completeness of nodes in the topology is not disturbed *i.e.,* once a node receives data-products, it continues to receive all the data-products that it requires. We first prove progress (Part 1) when N is added as a leaf. Then we prove progress (Part 2) even when N is added as a non-leaf node by *push* algorithm. In other words, Part 1 of proof proves that N receives all the data-products it requires, and Part 2 proves that once any node C starts receiving data-products, it continues to receive all the data-products it requires.

Part 1

The proof is by induction on the depth of the new node N that is added to the topology.

Base Case: The base case is trivial, where the depth of the new node N is 1. This means N is added as a child of the root Unidata. In this case, the control messages take only one hop (*i.e.,* the new node N requests the root R for the data-products). Let the set of these requested data-products be X. Since the root R has all the data-products, it starts sending all the requested data-products to N.

Inductive Hypothesis: Let us assume that the new node gets added at a

depth k. In this case, the control messages requesting data-products are sent from N upwards towards the root. These messages take maximum k hops and a finite time until they are satisfied. When the request in the control messages is satisfied, the requested data-products are sent downstream. The requested data-products take maximum k hops and finite time until they reach the node N.

As our inductive hypothesis, let us assume that each node in the topology at a depth $\leq k$ receives all the data-products it requires.

Proof for Inductive Step: Consider that the new node N gets added at a depth *k+1*. In this case, N requests the data-products it requires from its primary parent P at depth k. The parent P in turn sends the request for the data-products upwards towards the root. According to our inductive hypothesis, P (at a depth k gets these data-products within a finite time interval). Once P receives these data-products, it sends them downstream to N. Hence, the data-products take maximum *k+1* hops and a finite time interval to reach N.

Hence, we have shown that after the new node is added to the topology, it takes finite time interval for the new node to get the requested data-products.

Part 2

Now let us consider the case that the new node N is added by the *push* algorithm. Suppose node P adds N as its child and node C (the original child of P) becomes the child of N after executing the *push* algorithm. According to the algorithm shown in the Figure 10, P first adds N as its child. Then, the node N adds C as its child and finally the P-C link is cut-off (steps 1,2 and 3 from the algorithm *push*). As shown in Part 1, N starts receiving all the data-products it wants after a certain finite time interval. Also, when N adds C as a child node, C starts receiving data-products from N after a certain finite time interval. Only when C starts receiving data-products from N, P stops sending data-products to C. Hence, the data completeness of the

C in the topology is not disturbed. *i.e.,* in general all the other nodes continue to receive the data-products, they require.

Thus, assuming a non-faulty finite time interval after the new node N is joined to the topology, we have proved that the node N receives and all other nodes continue to receive the data-products they require. □

3. Tolerance

Lemma A.7. *A node crash causes many nodes in the topology to lose their node-disjointness. Consequently, rearrangements take place in the topology and these rearrangements restore the node-disjointness of all the affected nodes in the topology.*

Proof. Whenever a node crashes, its children lose their parent and its backed children lose their backup parent node. They may consequently lose node-disjointness. These nodes are called *affected nodes*. Due to the strong accuracy property of the perfect failure detector P, the affected nodes detect that the node has crashed. Consequently, they perform certain actions to regain their node-disjointness as shown below.

According to the Steps 2 and 3 from the algorithm to find a new parent in Figure 16, each child (say N) of the crashed node either

1. requests the root for a new parent as well as a new backup parent.
2. or requests its backup parent node to become its new primary parent.

In the first case, the child node N gets added to the topology as if it is a new node. So, it gets a new parent as well as a new backup parent. In the second case however, when the backup parent becomes new primary parent, N loses its backup parent. Consequently, N may lose node-disjointness, and it looks for another backup parent using the algorithm shown in Figure 17.

A backed child of the crashed node loses its backup parent. So it also may lose node-disjointness. It uses the strong accuracy property of the failure detector P and detects that its backup parent has crashed. It then looks for another backup parent using the algorithm shown in Figure 17.

After these actions are taken by the affected nodes, each node (except the direct children of the root) in the topology has a backup parent. We prove by induction that such changes restore the node-disjointness of the affected nodes in the topology.

The proof is by induction on the position of a node in the topology.

Base Case: Position of node = 1.

In this case, the node is at a depth 1 in the primary tree. These nodes are the children of the root and by definition they have node-disjointness.

Inductive Hypothesis: Position of node = *k*.

As our inductive hypothesis, we assume that all the nodes at position $\leq k$ satisfy node-disjointness.

Proof for Inductive Case: Position of node N = *k+1*.

Consider a node N at position *k+1* and let its parent node be P. There are two cases for the backup parent B of node N.

- *The backup parent B of N is not in the parent subtree of N.*

  The backup parent B is not in the subtree of P. By Definition A.6, the parent P is at a position *k*. So, by our inductive hypothesis, the parent P has node-disjointness. Hence, (by Lemma A.2) we prove that the node N has node-disjointness.

- *The backup parent node of N is in the parent subtree of N.*

  In this case, since the backup parent node is in the parent subtree of N, by Definition A.6, the backup parent B has the position value k and the parent

node P has position value $< k$. By inductive hypothesis, we know that both B and P have node-disjointness. Hence, from Lemma A.3 we prove that the node N has node-disjointness.

Thus, we show that after the rearrangements are done, each node in the topology (except the direct children of the root) has a backup parent. We also show that after the rearrangements are done, each node in the topology has node-disjointness. □

In the next chapter, we give the simulation results and the performance of our algorithms.

CHAPTER VIII

PERFORMANCE EVALUATION

In Chapter VI we presented heuristic algorithms to construct a distribution topology. The correctness of the algorithms is discussed Chapter VII. The correctness does not depend on the heuristic function. So if we use any heuristic function, the algorithms will be correct. On the other hand, the performance of the algorithms is dependent on the heuristic function. In this chapter, we evaluate the performance of our algorithms on the basis of comparison with the existing solution and a randomized algorithm which are explained later on in the chapter.

We have performed simulations of our algorithms on a DELL machine having 1.7 MHz processor speed with 512 MB RAM and running Fedora Core 2. We have used NS2 version 2.29 [12] to perform the simulations. To run these simulations, we needed complete information about the underlying topology. The information such as the out-bandwidth of a node, number of data-products (feed types) required by each node, name, IP address, and the domain of the server was obtained from the Unidata statistics in [28]. The size of each data-product is also obtained from these statistics.

To find the link latency of the unicast channel (i,j) we used the king tool [18]. This tool is developed by Krishna *et al.*, [10] and it gives an estimate of the RTT between any two arbitrary hosts in the Internet by estimating the RTT between their domain name servers. The tool depends on the fact that most of the domain name servers in current internet support recursive queries from any host in the Internet.

Since we did not have information about the bandwidth of each link, we made assumptions about the bandwidth. We assumed the bandwidth of each link to be 10 Mbps. If accurate information about the bandwidth of the links in the underlying

topology becomes available, then it can be used in the simulations instead of our assumptions.

For all the simulations, each end system is a representative of its domain. This means that we have not allowed two or more end systems from the same domain to be a part of the topology. The Unidata site is assumed to be the source for all the data-products. Our performance evaluation is based on characteristics such as the end-to-end latency of data reaching the servers, scalability of the algorithms, failure recovery, and automatic configuration of the LDM machines.

In these simulations, we evaluate the performance of our heuristic algorithms on basis of comparison with the existing solution, and comparison with a randomized solution. We designed a randomized solution which is similar to our heuristic solution, but it works in a random manner. In the randomized solution, a node R which is making decision of joining a new node N to the topology takes the following actions:

1. If R is not saturated, it joins the new node N.

2. Else if R is saturated, it randomly picks one of its children (say C_i, and throws N to C_i for addition.

3. If R throws N to C_i for addition, it backs the new node N (if R has enough resources to do so) or randomly picks one child C_j, (where $j \neq i$), and throws N to C_j for backup.

In the heuristic solution, a node R which is making decision of joining a new node N to the topology, may throw N to one of its children for addition and backup even if R is not saturated. In the randomized solution however, a decision making node R does not throw N to any of its children when it (the node R) is not saturated. Thus, heuristic approach attempts to create a tree with a height-width balance. On

the other hand, the randomized solution is a more greedy approach which attempts to create a flat tree with less depth.

A. End-to-End Latency

The end-to-end latency of the data reaching the nodes is the time taken for the data to reach the nodes after it leaves the Unidata distribution source. The evaluation of our algorithms with respect to the average end-to-end latency of data is done on the basis of comparison with the existing solution and the randomized solution in the following manner.

We calculate the average end-to-end latency of the data-products reaching the nodes in our topology. We then compare this latency with the average end-to-end latency of data at the nodes in the existing topology. We also compare this latency with the end-to-end latency of data at the nodes in the topology created by the randomized solution. To calculate the average end-to-end latency, we execute a three-step process:

1. Calculate the latency of data from parent node P to child node C for each parent-child pair (P,C).

2. Calculate the end-to-end latency for each node from the root (source of data distribution), as a sum of the link latencies obtained in Step 1.

3. Calculate the average of the end-to-end latencies at all nodes obtained from Step 2.

The latencies were calculated for a few sample data-products like HDS, CONDUIT, FNEXRAD, UNIWISC and FSL2. More information on these data-products can be obtained from [32]. These data-products were chosen because they are required by

most of the universities. To perform Step 1 (*i.e.,* to calculate the latency of data from parent to child for each parent-child pair), we performed NS2 simulations on our constructed topologies. After we had the results from Step 1, we calculated the end-to-end latency for each node (Step 2) by adding the latencies of all links in the primary route of that node from the source. In Step 3 we calculate the average of all the end-to-end latencies obtained from Step 2.

Consider the simple network topology shown in the Figure 20. In Step 1, we calculated the latencies of links such as latency of data from *R-R'*, *R'-A* and *R'-B*. In Step 2, we calculated the end-to-end latency of data from the root to all the nodes *R'*, *A*, and *B*. For example, the end-to-end latency of data at node *B* is the sum of latency(*R-R'*) and the latency(*R'-B*). Finally, we calculated the average end-to-end latency in Step 3 as an average of all the end-to-end latencies to nodes *R'*, *A*, and *B*.

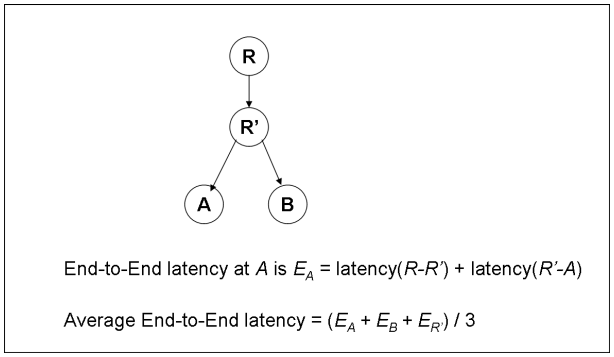

Fig. 20. Calculating Average End-to-End Latencies

Calculation of Results for the Existing Solution: The Unidata topologies for each data-product are available at [28]. We simulate these topologies in NS2 with the bandwidth assumptions mentioned before. Then we calculate the average end-to-end latency for all data-products using our three-step process.

Calculation of Results for Our Solution: From the Unidata topologies for each data-product, we find which universities require that data-product. We use the information about those universities to construct respective topologies for data distribution using our algorithm. We simulate the topologies in NS2 with the same bandwidth assumptions and then calculate the average end-to-end latencies for each data-product topology.

Calculation of Results for Randomized Solution: From the Unidata topologies for each data-product, we find which universities require that data-product. We use the information about those universities to construct respective topologies for data distribution using the randomized solution. We simulate these topologies in NS2 with the same bandwidth assumptions, and then calculate the average end-to-end latencies for each data-product topology.

Since the Unidata topologies are static, there is only one result (average latency) for each topology. But in our solution and the randomized solution, the order in which nodes are passed to the algorithm is a factor in the structure of the topology. So we perform 25 runs, each with a different order of nodes passed to the algorithm. We calculate the average end-to-end latency in each case, and we plot the mean of the results of 25 runs. We also plot the variance of the results of all 25 runs.

Figure 21 shows the comparison of average end-to-end latencies in our heuristic solution, the randomized solution, and the existing solution. From the graphs we can see that the average end-to-end latency of data delivery at the nodes in the randomized solution is less than that of our solution as well as the existing solution. We can also see that our solution gives a better performance with respect to end-to-end latency of data than Unidata's solution. The number of nodes in the topologies (simulated for the results in the above graph) is between 20 and 33. The randomized solution which

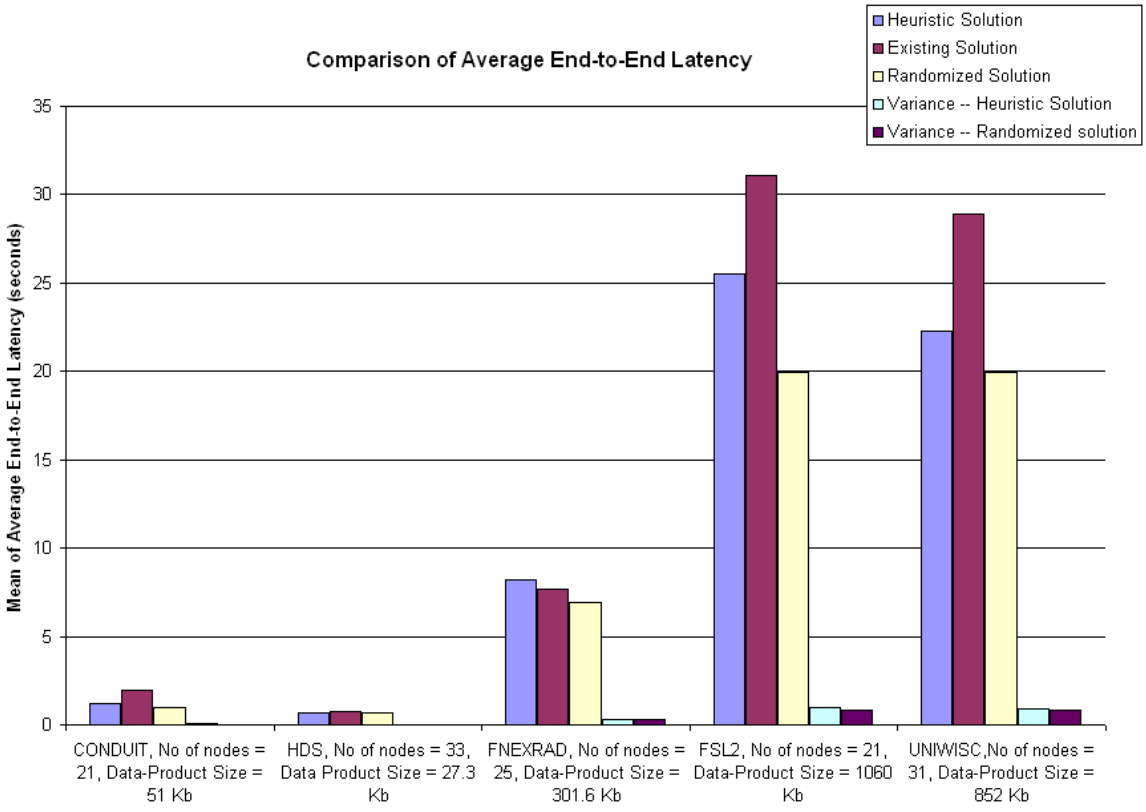

Fig. 21. Comparison of Average End-to-End Latencies

attempts to create a random and flat graph, performs better than our solution for these topologies. However, we will show in the next section that for topologies where the number of nodes in the graph is greater than 38 (approximately), our solution performs better than the random solution.

B. Scalability

We have used a distributed algorithm to construct the topology. Since the algorithm is distributed and does not depend on the complete knowledge of the topology, many nodes can be added to the topology at the same time provided they are being

processed at disjoint subtrees of the topology. The root of the topology can be a bottleneck if many nodes simultaneously request the root for addition. In the real-world scenario, this is somewhat rare[1]. Each node just keeps information about the nodes in its immediate neighborhood. So the extra overhead of keeping updated information about all the nodes in the topology is eliminated.

We perform simulations with increasing numbers of nodes in the topology and calculate the average end-to-end latency. We have information on 100 nodes from the Unidata statistics [28]. We chose independent sets of 20, 30, 40, 50, 70, 80, and 100 nodes and construct our topologies in each case. We then calculated the average end-to-end latencies for each topology using the same three-step process we used in the previous section. Each node sends data to its children nodes according to the data requirements of the children. The size of each data-product is also obtained from the Unidata website. So, for a parent-child pair (P, C), the data sent from P to C is the total size of all the data-products sent by P to C. For each point in the graph shown in Figure 22, we take 25 independent sets and 10-15 permutations of that set. We then plot the mean and the variance of all the calculated results. To compare the scalability of our solution and the randomized solution, we perform simulations on the topologies created by the randomized solution with the same input information. Figure 22 shows the variation in the average end-to-end latency of the data reaching the nodes, with increasing nodes in the topologies created by our solution as well as randomized solution.

The average end-to-end latency increases with increasing number of nodes. It is also evident from the graph that our solution is less efficient than the random solution for nodes less than 38 (approximately). However, as the number of nodes in the system

[1]The rate at which a new node gets added to the topology is around 1 per month as of May 2006.

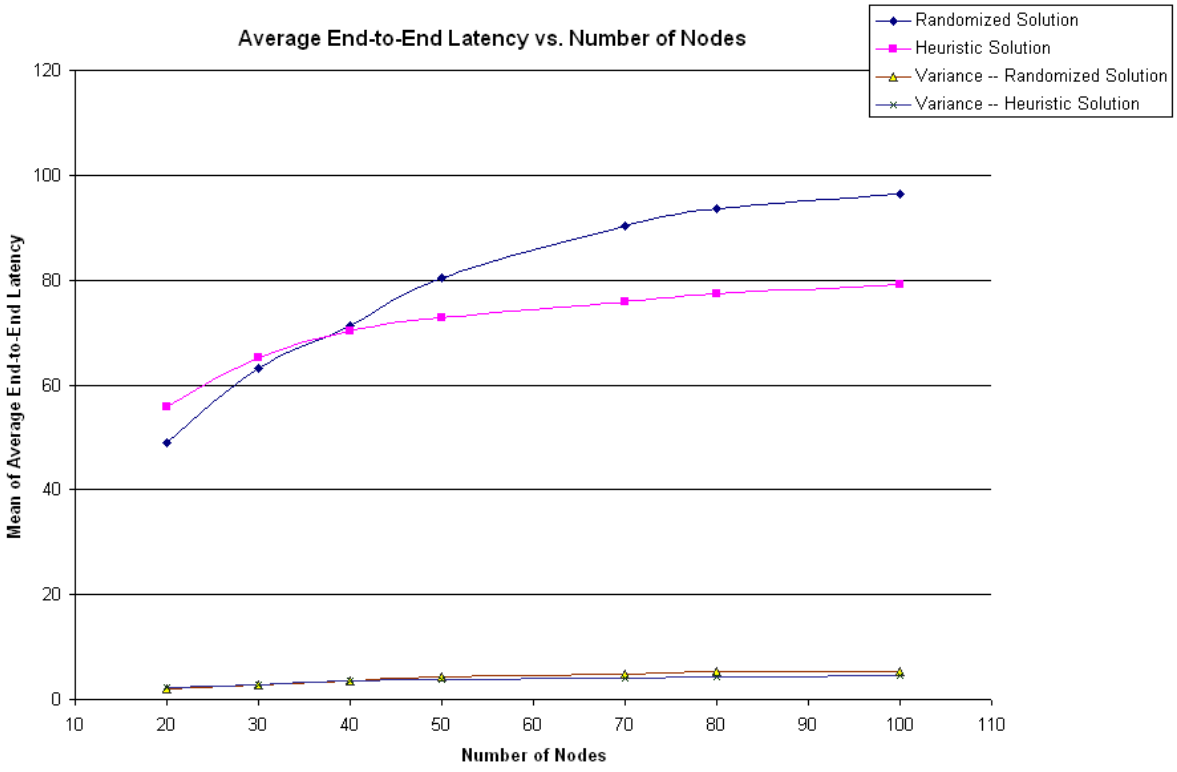

Fig. 22. Average End-to-End Latency vs. Number of Nodes

increases, our solution performs much better than the randomized solution.

As discussed before, the randomized solution attempts to create a flat tree. Thus in a graph constructed by a randomized solution, all the nodes (except the leaf nodes) are saturated. Due to this, whenever new nodes are added to the topology, they are always added as leaf nodes in lower levels of the topology. This increases the average end-to-end latency of data in the topology. Even if a new node is added to the topology as a non-leaf node (by the push operation), the tree becomes deeper and hence the average end-to-end latency of data increases.

On the other hand, in the topology constructed by heuristic solution, the non-

leaf nodes can be non-saturated. Hence, even if number of nodes in the topology increases, the topologies can facilitate addition of new nodes at upper levels in the tree. Thus, the average end-to-end latency of data does not increase much. When the new node is added as a non-leaf node however, the tree becomes deeper and the average end-to-end latency increases.

Thus, for fewer number of nodes in the topology the randomized solution creates a flat tree which has lower end-to-end latency; whereas the heuristic solution creates a balanced tree with a comparatively higher average end-to-end latency. As the number of nodes in the topology increases, the greedy approach of randomized solution creates inefficient trees and the average end-to-end latency increases by a large factor. The average end-to-end latency in the balanced tree created by the the heuristic solution also increases, but not by a major factor. The break-even point where both the curves intersect (Figure 22) is approximately 38.

C. Fault Tolerance

As described in Chapter VI, whenever a fault of a node P is detected by its backup parents and parent nodes, they stop sending data to P. The children and the backed children of P find another parent node and backup parent node respectively, for requesting data. Rearrangement of links take place in such a situation. According to Lemma A.7 after the rearrangements are done, each node in our constructed topology has node-disjointness. So, provided that the parent and backup parent nodes of a node N do not crash at the same time, all routes to N do not get cut at the same time and data delivery to N is not disrupted. However, the robustness is achieved at the expense of data redundancy and increased out-bandwidth usage at the nodes. Crashes cause rearrangement of links and changes in the topology. Ideally, since

crashes cause nodes to be removed from the topology, the average end-to-end latency should decrease. However, if the rearrangements are made in such a manner that the topology becomes deeper or inefficient, the average end-to-end latency could increase.

We performed simulations to show that when rearrangement of links occur in our solution, the average end-to-end latency decreases. We performed simulations with 50 nodes in the topology. The bandwidth assumption made in the simulations is 10 Mbps. We chose 20 independent sets, and for each set, we randomly chose 1, 2, 3, 5, 10, and 15 node crashes. We performed simulations on 10-15 permutations of each set. We calculated the average end-to-end latencies for each independent run and plot the mean and the variance of the results on the graph shown in Figure 23. In case of faults, if the leaf nodes crash, no rearrangement of links is done, and so only one link is removed from the topology. In this case, the average end-to-end latency must decrease. If the nodes in the top tier crash, many links are rearranged and the topology becomes flatter, thus reducing the average end-to-end latencies at the affected nodes.

From Figure 23, we can see that in case of zero faults the average end-to-end latency is around 73 seconds. However, as the number of faulty nodes increase, the average end-to-end latency decreases.

D. Automatic Configuration of the LDM

In our approach, new nodes join the topology automatically. When a new node wishes to be added to the topology, it sends a message to a controlling server, which then passes the control to the root of the topology. Control messages are sent among the nodes in the network and an appropriate node is chosen as a parent of the new node. An appropriate backup parent node is also chosen to back the new node. These nodes

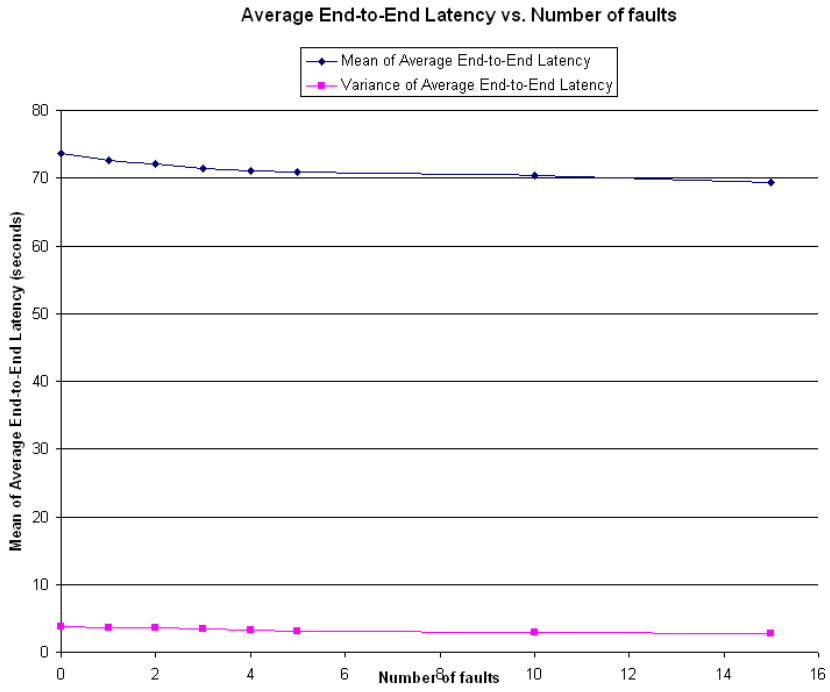

Fig. 23. Average End-to-End Latency vs. Number of Faulty Nodes

are chosen automatically by the algorithm and the LDM's of the sender nodes as well as the receiving nodes can be configured appropriately.

Once a node gets a new parent, backup parent, child or backed child node, it stores the information about this node in internal data structures. Every time the data structures are updated, these changes are reflected in the LDM configuration files[2], so that the corresponding servers can start the data transfer.

[2]The description about the LDM configuration files is beyond the scope of this thesis, however information about the configuration files can be found at [33]

CHAPTER IX

CONCLUSION, OPEN ISSUES AND FUTURE WORK

A. Conclusion

In this thesis, we have proposed efficient, scalable and distributed algorithms to construct an application-network for meteorological data distribution. Our algorithms are based on application layer multicast and they do not depend on the underlying network topology or the network routers for replicating, forwarding, or sending data to other end systems. We use a heuristic approach in our algorithms. To construct the distribution network, each server applies a heuristic function to make a decision based on local knowledge. This distribution topology forms the basis of the improved IDD system and the LDM.

We have constructed a decentralized but structured architecture for data distribution, which helps in reducing the end-to-end latency. The fan-out (and hence the bandwidth usage) of each node is bounded by the resource capacity of that node. Thus, our topology maintains a delay-bandwidth trade-off. Our algorithms handle topological changes dynamically, thus eliminating any form of manual intervention. Since the algorithms are based on local knowledge and local decisions, many nodes can be added to the topology concurrently. Moreover, the control overhead of keeping updated information about the complete topology is eliminated. The proposed algorithms are also fault-tolerant. The algorithms are designed to tolerate one crash at a time. They can tolerate multiple crashes at the same time, provided the topology is not partitioned due to simultaneous crashes. However, this robustness is achieved at the expense of data redundancy and increased usage of out-bandwidth at the nodes.

The performance of our algorithms depends on the heuristic function used in our

algorithms. As shown in Chapter VIII, the end-to-end latency of data delivery in our proposed algorithms is better than the end-to-end latency of data delivery in the existing solution. Also, if the number of nodes in the topology is greater than 38, our algorithms perform better than the algorithms of a randomized solution. This shows that our heuristic function is inefficient for smaller topologies. If the number of nodes in the system are less, our algorithms construct less efficient topologies than the topologies created by the randomized solution. This conclusion gives a stepping stone towards the improvement of heuristic function as well as our algorithms. Regarding fault tolerance, we showed that our topologies do not become inefficient or deeper as the number of crashed nodes increases.

The topologies created by the proposed algorithms will be used specifically for making improvements in Unidata's IDD system and the LDM and eventually for better meteorological data distribution.

B. Open Issues and Future Work

As discussed in the previous section, our overlays achieve low end-to-end latency as compared to the existing solution. However, for small numbers of nodes in the topology, our solution is not as efficient as the randomized solution. This is because the randomized solution constructs a random and a fairly flat tree, while our solution maintains a balance between the height and width of the tree for smaller topologies.

We have used a heuristic function in our algorithms. The function is not based on a theoretical analysis, but we measured its performance through simulation results. The structure of the constructed tree, depends on the heuristic function that we have used in the algorithms. Moreover, the correctness of our algorithms is independent of the heuristic function. So if we can find a better heuristic function, then we can

use the new heuristic function and our solution can become even better.

We have done a performance evaluation of the heuristic function using simulation results. However, we have not done a component based evaluation of the heuristic function. A component based evaluation of the heuristic function can give an analysis of all the five components of the heuristic and their impact on the structure of the topology. This evaluation can help us improve the heuristic function in the future.

Our algorithms are tolerant to one crash at a time. The algorithms can also tolerate multiple crashes, provided both crashes are such that the topology is not partitioned. Our solution can tolerate multiple crashes if the parent and the backup parent nodes of a particular node, do not crash at the same time. However, this robustness is achieved at the expense of data redundancy and increased usage of out-bandwidth at the nodes. If the application requires more tolerance, so as to tolerate multiple crashes (without any restrictions on the crashed nodes), we need to provide multiple backup parents rather than a single backup parent. This decision involves making a trade-off and allocating more bandwidth to the nodes to support more backed children.

Each node in the topology has to reserve enough bandwidth to send all the K elements of the global set of data-products to a downstream node. This extra bandwidth is reserved for the push algorithm, where a parent node pushes a new node, between itself and its child node. In this case, the parent node temporarily supports both the new node and the child node, so as to maintain data-completeness for the child node. However, the surplus bandwidth is not used anytime except the push algorithm. So, if the application in the future finds that the push operation is infrequent, then Unidata can reserve enough bandwidth to send all K elements of the global set of data-products. It can then support the child node temporarily, so as to maintain the child node's data-completeness. In this case, Unidata is the

only node that can temporarily support a child node during push operation. This creates a bottleneck for adding nodes by push operation and reduces the scalability of the algorithms. Another implication is that only one crash in the topology can be tolerated at any time.

We have used a perfect failure detector for fault detection. A perfect failure detector is the strongest of all the failure detectors. In the future, we would like to design or modify our algorithms so that they can use a weaker failure detector like $\Diamond P$. This detector is called an eventually perfect failure detector. $\Diamond P$ has strong completeness property (which is same as the completeness property of the perfect failure detector P). However, it has a weaker accuracy property. This property is called eventually strong accuracy and it is stated as, "There is a time after which correct processes are not suspected by any correct process". This property means that the failure detector $\Diamond P$ can make finitely many mistakes, and after a time it accurately detects crash faults.

So if we can modify the algorithms in such a manner that they can tolerate the finitely many mistakes of the eventually perfect failure detector $\Diamond P$, then we can use the weaker oracle. The advantage of using a weaker oracle is that it is easy to implement than a stronger oracle. One idea of using the $\Diamond P$ detector, is to ignore the suspect list given by the failure detector for a certain fixed number of steps or a fixed time. For example, consider a node N running our algorithms and its neighbor N'. The neighbor can be a parent, backup parent, child or a backed child node. If N running learns that the failure detector $\Diamond P$ has suspected N', then it waits for α rounds of execution [1] (where α is fixed). Suppose that the failure detector $\Diamond P$

[1] A round of execution is action taken by a node, in which it receives data-products from parent, stores them in the product-queue and then sends the data-products to child node.

suspects neighbor N' in all the α rounds. Then, N assumes that $\Diamond P$ has converged and it concludes that N' has really crashed. However, if $\Diamond P$ does not suspect N' in all the α rounds, then N does not assume that $\Diamond P$ has converged and does not conclude that N' has crashed. In such a manner, a node N can make conclusions about the crash of its neighbors and take appropriate actions.

We have performed simulations of our algorithm to evaluate its performance. However, we would like to implement these algorithms, in real world with real data-products. Another idea is to implement these algorithms using PlanetLab [20].

We have not solved the problem of large product-queues mentioned in the Section D of Chapter III. In the future we would like to design algorithms that will reduce the product-queue size as well.

REFERENCES

[1] M. S. Baltuch, Unidata's Internet Data Distribution (IDD) System: Two Years of Data Delivery, presented at the *13th International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography and Hydrology*, American Meteorological Society, Long Beach, CA, February 1997.

[2] S. Banerjee, B. Bhattacharjee and C. Kommareddy, Scalable Application Layer Multicast, *Proc. ACM SIGCOMM*, 205–217, Pittsburgh, PA, August 2002.

[3] T. D. Chandra and S. Toueg, Unreliable Failure Detectors for Reliable Distributed Systems, *JACM: Journal of the ACM*, 43(2), 225–267, March 1996.

[4] S. M. Chiswell, Towards Real-Time Collection of Internet Data Distribution Statistics for Automated Topology and Performance Monitoring, presented at the *18th International Conference on Interactive Information and Processing Systems (IIPS) for Meteorology, Oceanography and Hydrology*, American Meteorological Society, Orlando, FL, Janurary 2002.

[5] Y. Chu, S. G. Rao and H. Zhang, A Case for End System Multicast, *Proc. ACM SIGMETRICS*, 1–12, Santa Clara, CA, June 2000.

[6] S. Deering, Host Extensions for IP Multicasting, RFC 1112, Stanford University, CA, August 1989.

[7] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei, Protocol Independent Multicast -Sparse Mode (PIM-SM): Protocol Specification, RFC 2362, Internet Engineering Task Force (IETF), June 1998.

[8] P. Francis, Yoid: Your Own Internet Distribution, available online at http://www.aciri.org/yoid/ April 2000.

[9] M. R. Garey and D. S. Johnson, *Computers and Intractibility: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979.

[10] K. P. Gummadi, S. Saroiu and S. D. Gribble, King: Estimating Latency between Arbitrary Internet End Hosts, *ACM SIGCOMM Internet Measurement Workshop (IMW)*, 5–18, Marseille, France, November 2002.

[11] D. A. Helder and S. Jamin, End-Host Multicast Communication Using Switch-Trees Protocols, *Proc. 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, 419–424, Berlin, Germany, May 2002.

[12] Information Sciences Institute, http://www.isi.edu/nsnam/ns/ns-build.html Introduction and Links for Downloading, Installing, and Running the Network Simulator. Accessed on June 2006.

[13] Internet2 NetFlow, http://netflow.internet2.edu/weekly/20060501/ Statistics of the Internet2 Usage by Various Applications. Accessed on May 1, 2006.

[14] J. Könemann, Approximation Algorithms for Minimum-Cost Low-Degree Subgraphs, Ph.D dissertation, Carnegie Mellon University, Pittsburg, PA, 2003.

[15] M. Kwon and S. Fahmy, Topology-Aware Overlay Networks for Group Communication, *Proc. of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 127-136, Miami, FL, 2002.

[16] Q. Lv, P. Cao, E. Cohen, K. Li and S. Shenker, Search and Replication in Unstructured Peer-to-Peer Networks, *Proc. 16th International Conference on*

Supercomputing (ICS), 84–95, New York, June 2002.

[17] N. Malouch, Z. Liu, D. Rubenstein and S. Sahu, A Graph Theoretic Approach to Bounding Delay in Proxy-Assisted, End-System Multicast, *Proc. of the Tenth IEEE International Workshop on Quality of Service (IWQoS)*, 106-115, Miami Beach, FL, May 2002.

[18] Max Planck Institute for Software Systems, http://www.mpi-sws.mpg.de/ gummadi/king/ This web page gives Information about the King Tool. Accessed on June 2006.

[19] J. Moy, Multicast Extensions to OSPF, RFC 1584, Internet Engineering Task Force (IETF), 1994.

[20] PlanetLab, http://www.planet-lab.org/ Introduction and Information about PlanetLab. Accessed on June 2006.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker, A Scalable Content-Addressable Network, *Proc. of ACM SIGCOMM*, 161-172, San Diego, CA, August 2001.

[22] R. Ravi, M. V. Marathe, S. Ravi, D. J. Rosenkrantz, H. B. Hunt III, Approximation Algorithms for Degree-Constrained Minimum-Cost Network-Design Problems, *Algorithmica*, 31(1), 58-78, 2001.

[23] S. Shi and J. Turner, Multicast Routing and Bandwidth Dimensioning in Overlay Networks, *IEEE Journal on Selected Areas in Communications*, 20(8), 1444-1455, October 2002.

[24] S. Tan, A. G. Waters and J. S. Crawford, MeshTree: A Delay optimised Overlay Multicast Tree Building Protocol, Tech. Report 5-05, University of Kent,

University of Kent, April 2005.

[25] University Center for Atmospheric Research, http://www.unidata.ucar.edu/software/ldm/ Overview of the Unidata's Local Data Manager (LDM) Software. Accessed on April 2006.

[26] University Center for Atmospheric Research, http://www.unidata.ucar.edu/software/idd/ Overview of the Unidata's Internet Data Distribution (IDD). Accessed on April 2006.

[27] University Center for Atmospheric Research, http://www.unidata.ucar.edu/software/ldm/ldm-6.4.5/basics/glindex.html Glossary for Keywords Related to LDM 6.4.5. Accessed on April 2006.

[28] University Center for Atmospheric Research, http://www.unidata.ucar.edu/software/idd/rtstats/ Detailed Statistics of the Servers Participating in the Unidata IDD. Accessed on April 2006.

[29] University Center for Atmospheric Research, http://www.unidata.ucar.edu/software/ldm/ldm-6.4.5/basics/generic-LDM.html Information about the Generic Run Time Structure of the LDM 6.4.5. Accessed on April 2006.

[30] University Center for Atmospheric Research, http://www.unidata.ucar.edu/ Information about the Unidata Program Center. Accessed on April 2006.

[31] University Center for Atmospheric Research, http://www.unidata.ucar.edu/software/idd/currentStatusOverview.html

Overview and Statistics of Status of the Internet Data Distribution. Accessed on May 2002.

[32] University Center for Atmospheric Research, http://www.unidata.ucar.edu/software/ldm/ldm-current/basics/feedtypes/index.html Information about the Feed Types Used in Unidata's LDM Software. Accessed on April 2006.

[33] University Center for Atmospheric Research, http://www.unidata.ucar.edu/software/ldm/ldm-6.4.5/basics/configuring.html Detailed description of Installation and Configuration of an LDM. Accessed on May 25, 2006.

[34] D. Waitzman, C. Partridge, and S. E. Deering, Distance Vector Multicast Routing Protocol, RFC 1075, Internet Engineering Task Force (IETF), November 1988.

[35] Z. Wang and J. Crowcroft, Bandwidth-Delay Based Routing Algorithms, *IEEE GLOBECOM*, 3, 2129–2133, Singapore, November 1995.

[36] B. Zhang, S. Jamin, L. Zhang, Host Multicast: A Framework for Delivering Multicast to End Users, *Proc. IEEE INFOCOM*, 1366–1375, New York, June 2002.

VITA

Name	*Saurin Bipin Shah*
Education Background	*B.E., Mumbai University*, 2004
Mailing Address	16643, *29th Way NE*, *Apt N* 2042, *Redmond*, *WA* 98052

The typist for this thesis was Saurin Bipin Shah.